¡¡APRUEBA TU EXAMEN CON SCHAUM!!

Programación en Java 2

Schaum

Jesús Sánchez Allende • Gabriel Huecas Fernández-Toribio • Baltasar Fernández Manjón
Pilar Moreno Díaz • Antonio José Reinoso Peinado • Ricardo Sosa Sánchez-Cortés

- REDUCE TU TIEMPO DE ESTUDIO
- MÉTODO DE APRENDIZAJE PROGRESIVO PARA PROGRAMAR EN JAVA
- EJERCICIOS RESUELTOS COMENTADOS CON REFERENCIAS EN LA TEORÍA
- TRATA LA VERSIÓN 5.0 DE JAVA 2 Y ANTERIORES

zalo para las siguientes asignaturas:

- NDAMENTOS DE PROGRAMACIÓN
- ☑ LABORATORIO DE PROGRAMACIÓN
- DESARROLLO ORIENTADO A OBJETOS
- ☑ METODOLOGÍA DE LA PROGRAMACIÓN

PROGRAMACIÓN EN JAVA 2

PROGRAMACIÓN EN JAVA 2

Jesús Sánchez Allende
Doctor Ingeniero de Telecomunicación
Dpto. de Electrónica y Sistemas
Universidad Alfonso X El Sabio

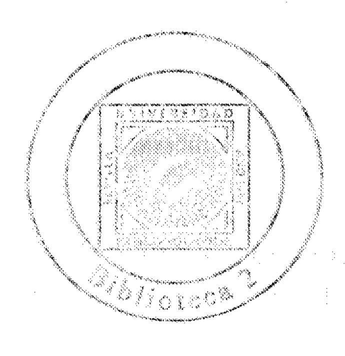
Gabriel Huecas Fernández-Toribio
Doctor Ingeniero de Telecomunicación
Dpto. de Ingeniería de Sistemas Telemáticos
Universidad Politécnica de Madrid

Baltasar Fernández Manjón
Doctor en Ciencias Físicas
Dpto. de Sistemas Informáticos y Programación
Universidad Complutense de Madrid

Pilar Moreno Díaz
Licenciada de Matemáticas
Dpto. de Electrónica y Sistemas
Universidad Alfonso X El Sabio

Antonio José Reinoso Peinado
Ingeniero en Informática
Dpto. de Electrónica y Sistemas
Universidad Alfonso X El Sabio

Ricardo Sosa Sánchez-Cortés
Ingeniero en Informática
Dpto. de Electrónica y Sistemas
Universidad Alfonso X El Sabio





MADRID • BOGOTÁ • BUENOS AIRES • CARACAS • GUATEMALA • LISBOA • MÉXICO
NUEVA YORK • PANAMÁ • SAN JUAN • SANTIAGO • SÃO PAULO
AUCKLAND • HAMBURGO • LONDRES • MILÁN • MONTREAL • NUEVA DELHI • PARÍS
SAN FRANCISCO • SIDNEY • SINGAPUR • ST. LOUIS • TOKIO • TORONTO

Programación en Java 2. Serie Schaum



McGRAW-HILL/INTERAMERICANA DE ESPAÑA, S. A. U.
Edificio Valrealty, 1ª planta
Basauri, 17
28023 Aravaca (Madrid)

www.mcgraw-hill.es
universidad@mcgraw-hill.com

ISBN: 84-481-4591-7
Depósito legal: M. 24.998-2005

Editor: Carmelo Sánchez González
Compuesto en: GAAP Editorial, S. L.
Impreso en:

IMPRESO EN ESPAÑA - PRINTED IN SPAIN

A mis padres y a mis chicas, por estar conmigo.
JSA

A mi mujer, Paqui
y a mis hijos, Fernando, Carmen y Raquel.
GHFT

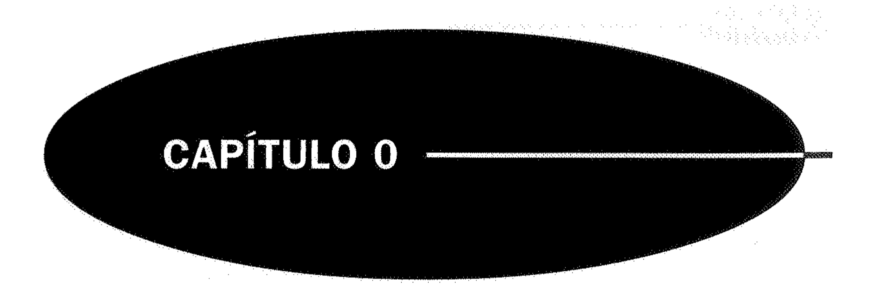
A Pilar, Balti y Celia.
BFM

A mis padres.
PMD

A mis padres, Luis y María.
AJRP

A mis padres Gloria y Ricardo por apoyarme siempre en todo.
RSS

CONTENIDO



¿Qué es Java?

0.1 EL LENGUAJE DE PROGRAMACIÓN JAVA Y LA PLATAFORMA JAVA

El lenguaje de programación Java es un lenguaje moderno, presentado por primera vez por Sun Microsystems en el segundo semestre de 1995. Desde el principio ganó adeptos rápidamente por muy diversas razones, una de las más importantes es su neutralidad respecto de la plataforma de ejecución lo que permite, entre otras cosas, añadir programas a una página Web.

Pero quizá lo que más guste a los programadores son un par de aspectos que le hacen muy cómodo y agradable de usar para programar:

- La sencillez y elegancia de cómo se escriben los programas en Java. A ello se une que es un lenguaje orientado a objetos que evita muchas preocupaciones a los programadores. En el proceso de compilación se realizan multitud de comprobaciones que permiten eliminar muchos posibles errores posteriores.
- Las bibliotecas ya definidas que proporciona el lenguaje y que el programador puede utilizar sin tener que hacerlas de nuevo.

La evolución de Java ha sido muy rápida. Desde que se hizo público el lenguaje y un primer entorno de desarrollo, el JDK (Java Development Kit), hasta el momento actual, la plataforma Java ha ido creciendo constantemente y a un ritmo cada vez mayor según se han ido incorporando un gran número de programadores de todo el mundo.

Pero Java 2 no es sólo un lenguaje. Es una plataforma de desarrollo de programas que consta de:

- Un lenguaje de programación: el lenguaje Java, del mismo nombre que la plataforma.
- Un conjunto de bibliotecas estándar que se incluyen con la plataforma y que deben existir en cualquier entorno con Java. También se denomina Java Core. Estas bibliotecas comprenden: strings, procesos, entrada y salida, propiedades del sistema, fecha y hora, Applets, API de red, Internacionalización, Seguridad, Componentes, Serialización, acceso a bases de datos, etc.
- Un conjunto de herramientas para el desarrollo de programas. Entre ellas cabe citar el compilador de Java a código de bytes, el generador de documentación, el depurador de programas en Java, etc.
- Un entorno de ejecución cuyo principal componente es una máquina virtual para poder ejecutar los programas en código de bytes.

La plataforma Java2 se puede utilizar desde distintos sistemas operativos, ejecutándose cada uno de ellos en el hardware correspondiente. (Véase Figura 0.1.)

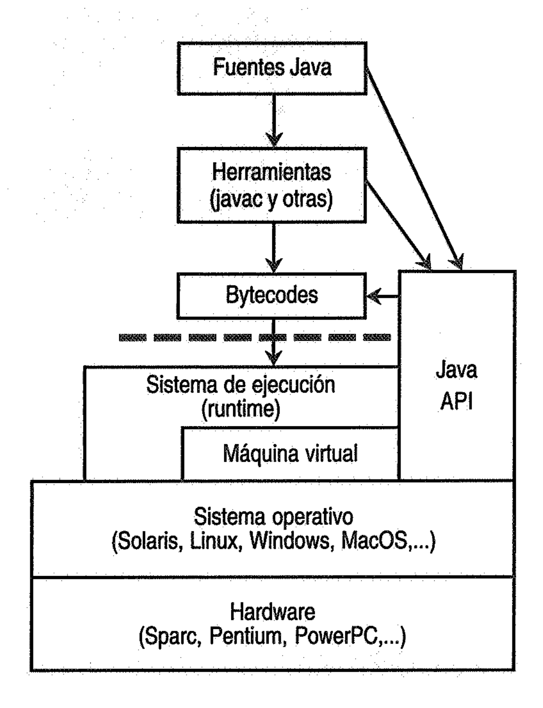


Figura 0.1. Elementos de la plataforma Java 2.

0.2 DESARROLLO DE PROGRAMAS

El desarrollo de programas en Java, al igual que ocurre normalmente en otros lenguajes de programación, sigue un proceso como el siguiente:

- **Edición del programa fuente.** Se denomina programa fuente al programa que se escribe utilizando un entorno de programación como los descritos, o cualquier editor, en un lenguaje de programación. En Java los archivos fuentes tienen un nombre como nombre.java, donde el nombre ha de ser el mismo que el empleado como nombre de la clase y la extensión ha de ser .java.
- **Compilación.** Tras escribir el programa hay que compilarlo utilizando un compilador. Los entornos de desarrollo facilitan esa tarea, haciéndola casi invisible en alguno de ellos. También se puede compilar utilizando la línea de comandos con la herramienta javac, de la siguiente forma: javac nombre.java.

  En el proceso de compilación se comprueba que lo que se ha escrito es correcto en Java y se traduce a otro lenguaje, denominado código de bytes (bytecode). Si durante la compilación se detectan errores, el entorno avisará de los problemas detectados y dónde se han encontrado para que pueda corregirlos. Si en la compilación no se detecta ningún error se genera un archivo como nombre.class, con el mismo nombre que la clase que se compila pero con la extensión .class.
- **Ejecución.** Una vez compilado se ejecuta el programa y se comprueba si hace lo que se había previsto. Si el programa no hace lo previsto se vuelve a editar, modificando los aspectos que no funcionan adecuadamente.

0.3 ASPECTOS SOBRE CÓMO ESCRIBIR LOS PROGRAMAS

Los pasos indicados anteriormente dan una idea muy breve de cómo escribir programas. Cuando escriba un programa debe tener en cuenta a qué dar más importancia de acuerdo con la siguiente máxima:

Legibilidad > Corrección > Eficiencia

Que viene a indicar la importancia relativa que debe conceder a estos tres aspectos en la programación:

- **Legibilidad**: El programa ha de ser fácil de leer y entender, incluso para una persona que no haya participado en el desarrollo del programa. Este aspecto es en la actualidad el más importante. A pesar de lo que pueda pensar en este momento, dar legibilidad a un programa hará que los otros aspectos salgan ganando.
- **Corrección**: Un programa debe hacer lo que tiene que hacer, ni de más, ni de menos. Se supone que con la fase de pruebas se comprueba hasta cierto nivel que es cierto y que el programa funciona correctamente.
- **Eficiencia**: Suele ser una preocupación típica de muchos programadores. La eficiencia se suele medir en tiempo que se tarda en ejecutar o en cantidad de memoria que ocupa el programa. Nuestro consejo es que se olvide completamente de este tema. Una vez domine el lenguaje ya tendrá tiempo de ocuparse de ello.

En el Apéndice B puede encontrar recomendaciones específicas para mejorar la legibilidad de un programa y, por tanto, debería conocerlas y seguirlas lo máximo posible. Siempre que pueda fíjese en cómo escriben los programas los buenos programadores.

0.4 ENTORNOS DE DESARROLLO

Existen multitud de fabricantes que disponen de entornos de desarrollo para Java. En primer lugar están teniendo una gran aceptación algunos entornos de libre distribución. Puede utilizar otros entornos, aunque sean de pago, ya que existen versiones reducidas que se pueden utilizar para aprender los fundamentos del lenguaje y la programación con Java.

En esta sección se va a hacer un recorrido rápido por cuatro entornos de desarrollo en Java que puede obtener de forma gratuita y cómo utilizarlos para las tareas básicas que, como se han descrito en la sección anterior, son:

- Editar el programa fuente.
- Compilación y ejecución del programa.

Dada la extensión del libro, no se describirán otros aspectos de los entornos de programación. Tenga en cuenta también, que los entornos que aquí se describen son las últimas versiones disponibles de los programas en el momento final de entrega del libro manuscrito.

Dada la evolución de los entornos es fácil que la versión actualizada que usted pueda encontrar ya haya variado. Sin embargo, esperamos que le siga siendo útil esta presentación.

0.4.1 NetBeans (*java.sun.com* o *www.netbeans.org*)

NetBeans es un proyecto de creación de un entorno de libre distribución profesional patrocinado por Sun MicroSystems. En java.sun.com puede encontrar un paquete completo que viene con el entorno de desarrollo NetBeans y la última versión del lenguaje. Puede encontrarlo en *java.sun.com/j2se/1.5.0/*

Tras iniciar NetBeans aparece una ventana como la de la Figura 0.3. Para empezar se elige en ella File -> New Project. En la ventana que aparece seleccione Java Aplication. A continuación, aparece la ventana de la Figura 0.2. En ella introduzca en Project Name el nombre del proyecto, por ejemplo Notas. En el último cuadro de texto (Create Main Class) sustituya el nombre Main que aparece por defecto por el nombre de la clase principal que desee, en este caso Notas, como ya aparece en la Figura 0.2.

En la ventana de trabajo aparece parte de una clase ya realizada. En ella debe completar el método main. También puede eliminar todo lo que no va a necesitar para su aplicación. Una vez terminada de escribir la clase puede compilarla y ejecutarla en un paso pulsando sobre la flecha verde hacia la derecha de la barra de herramientas de la parte superior.

En la parte inferior se mostrará tanto el proceso de compilación y los errores que pueda tener el programa como el resultado de la ejecución del mismo.

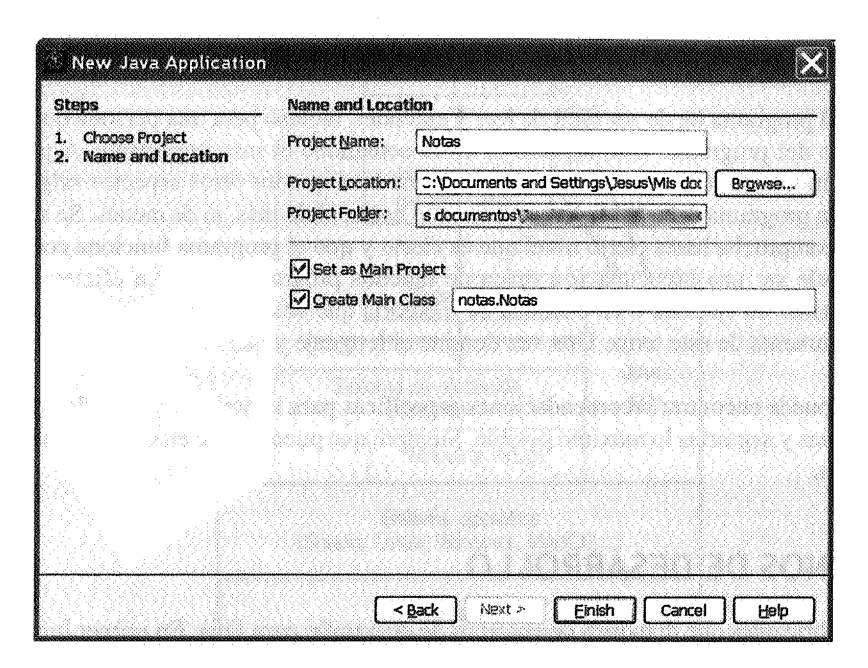


Figura 0.2. Ventana de configuración de proyecto de NetBeans.

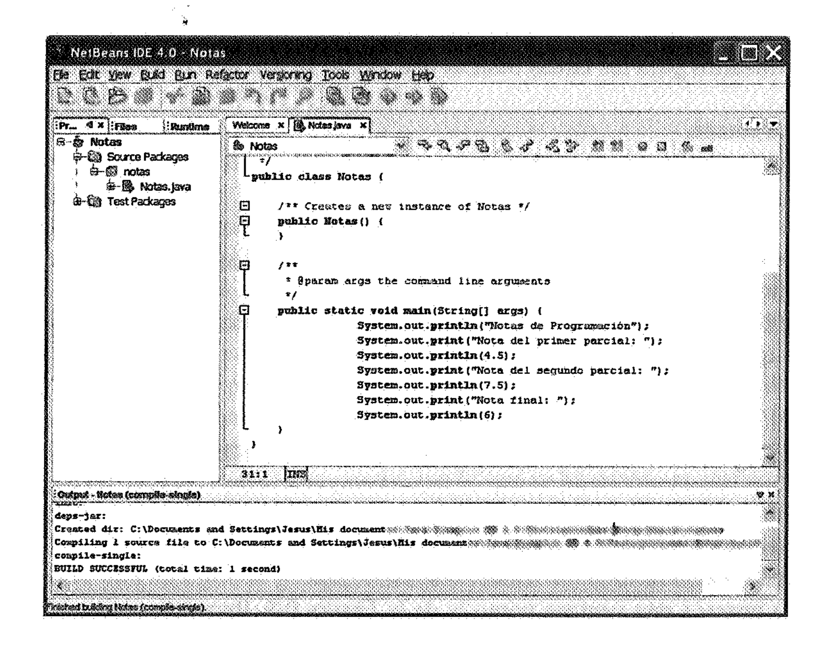


Figura 0.3. Ventana principal de trabajo de NetBeans.

0.4.2 Eclipse (*www.eclipse.org*)

Eclipse es un proyecto de creación de un entorno genérico de desarrollo patrocinado principalmente por IBM. Su objetivo es crear un marco de trabajo para distintas tareas y lenguajes de programación. Se puede encontrar una versión preparada para trabajar con Java en *www.eclipse.org/downloads/*. Tras iniciar el programa para empezar un nuevo proyecto seleccione File -> New -> Project. Aparece la ventana de la Figura 0.4. En ella seleccione Java Project.

A continuación, en la ventana siguiente introduzca en el campo de texto Project Name el nombre del proyecto, por ejemplo Notas. Tras generar el proyecto pulse Finish para terminar.

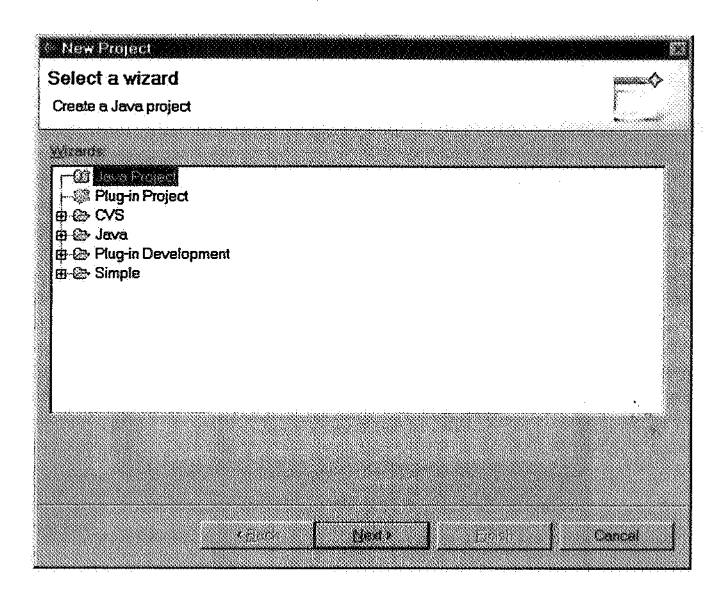


Figura 0.4. Ventana de selección de un nuevo proyecto en Eclipse.

Una vez generado el proyecto aparece en la parte derecha del entorno de desarrollo la información del proyecto. Para crear la clase inicial del programa seleccione, pulsando con el botón derecho del ratón sobre el nombre del proyecto, la opción New -> Class, y en la ventana que aparece rellene los campos Name, con el nombre de la clase, por ejemplo Notas y marque la casilla de verificación para crear el método public static void main(), tal como aparece en la Figura 0.5.

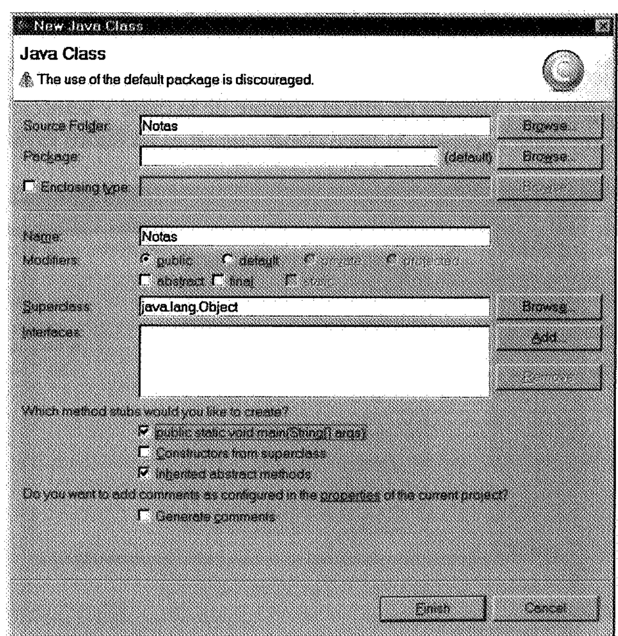


Figura 0.5. Ventana de creación de la clase principal en Eclipse.

Una vez aparezca el código fuente generado complételo como aparece en el Figura 0.6. Para ejecutarlo seleccione el icono verde con una flecha hacia la derecha. Le aparecerá una ventana de configuración de la ejecución la primera vez. En ella seleccione en la parte derecha Java Aplication, seleccione en el modo Run la opción Java y haga doble clic en Java Aplication para, por último hacer doble clic en Notas.java. Las siguientes veces ya guarda esta configuración y basta con hacer clic en el icono verde de ejecución.

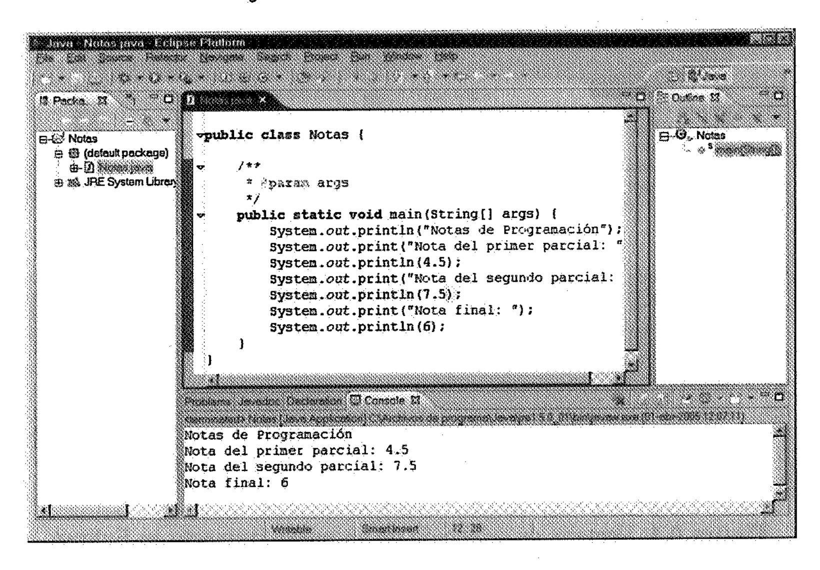


Figura 0.6. Ventana de trabajo de Eclipse.

0.4.3 BlueJ (*www.bluej.org*)

BlueJ es un entorno desarrollado en la Universidad de Kent. Antes de instalar el entorno debe descargar de la página de Sun el paquete J2SE e instalarlo. Como podrá observar en las siguientes figuras, su aspecto es mucho más sencillo que los entornos anteriores.

Al iniciarse debe elegir Project -> New Project y elegir un nombre para el proyecto, por ejemplo Notas. A continuación, aparece la pantalla de la Figura 0.7. Escriba en ella el nombre de la clase y seleccione Class. Se creará un recuadro rayado (significa que no está compilada la clase). Si hace doble clic en el recuadro rayado podrá editar la clase.

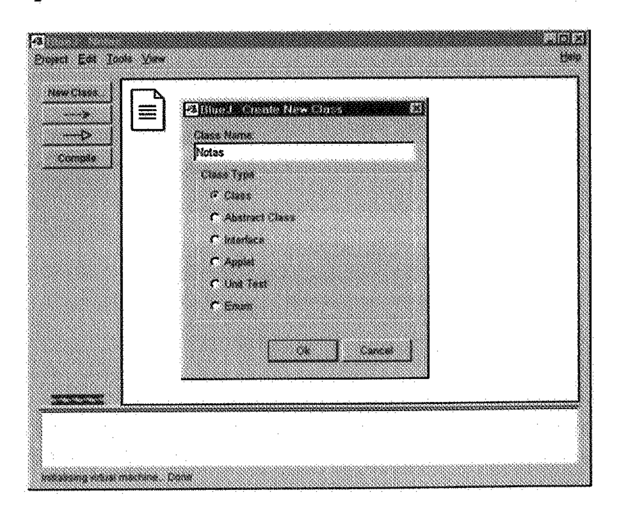


Figura 0.7. Pantalla de creación de una nueva clase, interface, enumerado, etc.

Por defecto se crea una clase con muchos elementos predefinidos. Puede borrar prácticamente todo dejando sólo el nombre de la clase y las llaves de apertura y cierre de la misma. Escriba el método main y el resto de la clase como en el ejemplo de la Figura 0.8. Para compilar, pulse el botón Compile de la ventana de edición o de la ventana de clases. Una vez compilado el programa, si compila correctamente, el recuadro que representa la clase aparecerá sin rayar. Para ejecutar el programa pulse con el botón derecho sobre él y seleccione el método main. Pulse OK. Aparecerá una nueva ventana con el resultado de la ejecución.

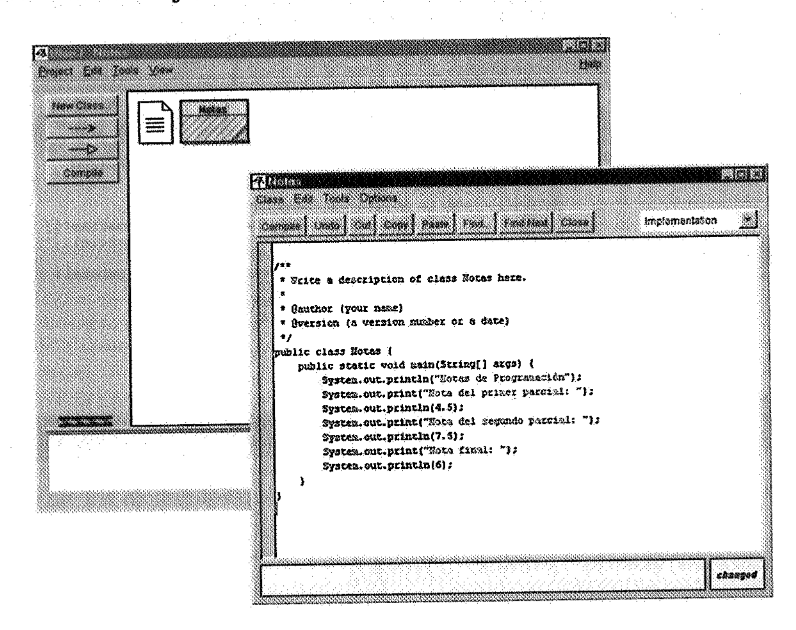


Figura 0.8. Ventana general y de edición de clases de BlueJ.

0.4.4 JCreator LE (*www.jcreator.com*)

La versión LE es una versión de uso gratuito. Tras indicar su nombre y dirección de correo se le enviará un URL de donde descargarse el programa. Tras instalarlo puede seleccionar File -> New -> New project. Aparece la ventana de la Figura 0.9 donde puede dar un nombre al proyecto. En este caso se le ha dado el nombre Notas.

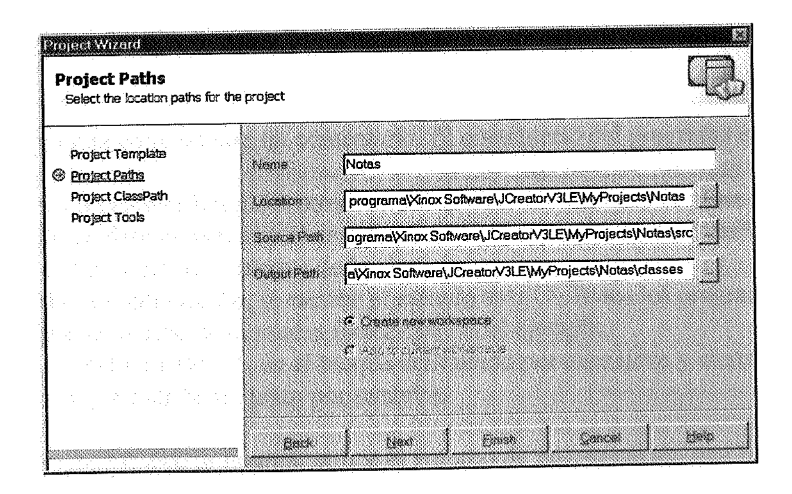


Figura 0.9. Ventana de creación del proyecto en JCreator.

Aparece entonces la ventana principal como la de la Figura 0.9. Haga doble clic en el nombre del programa Notas.java y borre lo que no necesite del código que aparece en la ventana de trabajo. Pulse el botón de compilar y después el de ejecutar (como se ha marcado en la Figura 0.10). Al ejecutar el programa aparece una ventana con el resultado de la ejecución.

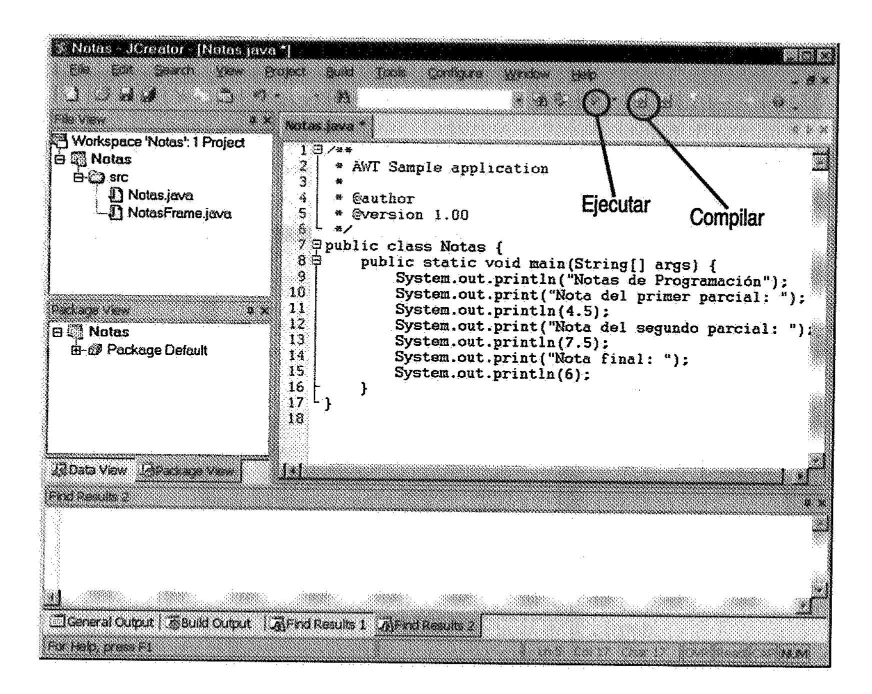
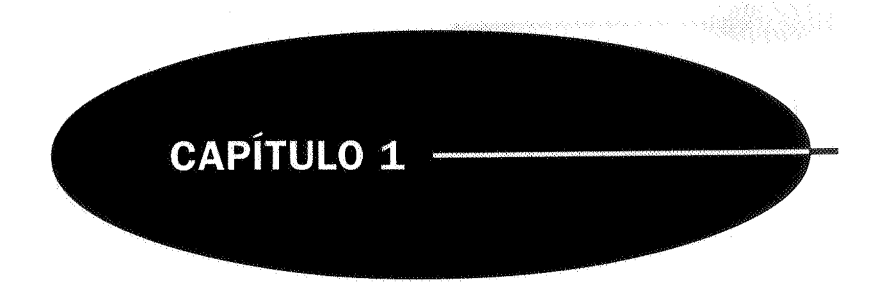


Figura 0.10. Pantalla principal de trabajo de JCreator.



Introducción a la programación

1.1 ESTRUCTURA DE UN PROGRAMA

Un programador debe utilizar los elementos que le ofrece el lenguaje de programación para, utilizándolos de forma inteligente y lógica, crear un programa que resuelva un problema.

En el siguiente programa de ejemplo puede ver un programa muy sencillo escrito en Java. Este programa escribe en la pantalla el texto Hola a todos.

```
/**
 * Programa en Java que escribe un texto en la pantalla.
 */
public class Hola {
    public static void main(String[] args) {
        System.out.println("Hola a todos.");
    }
}
```

En este programa se pueden ver algunos de los elementos que se van a utilizar siempre que se escribe un programa en Java:

- **Comentario.** El programa empieza con un comentario. El comentario del programa empieza con /** y acaba con */.
- **Definición de clase.** La primera línea del programa después del comentario define una clase que se llama `Hola`. La definición de la clase empieza en el carácter abre llave { y termina en el carácter cierra llave }. Todo lo encerrado entre las llaves define el bloque de la clase.
- **Definición de método.** A continuación, se escribe el método `main()`. Todos los programas en Java deben tener un método `main()`, que se escribe de la misma forma que el del ejemplo.
- **Sentencias.** Dentro del método `main()`, en el bloque delimitado por abre llave y cierra llave, existe una única sentencia o instrucción que escribe un texto por pantalla.

Para ver el resultado de ejecutar este programa, utilice el entorno de programación que haya elegido de la forma descrita en la documentación del mismo. Tenga en cuenta que en Java las letras mayúsculas y las letras minúsculas son letras distintas, por lo que debe escribirlas tal como aparecen en el ejemplo dado.

La estructura general de una clase principal en Java suele ser la siguiente:

```
/**
 * Estructura de una clase en Java
*/
public class NombreDeClase {
    // Declaración de los atributos de la clase

    // Declaración de los métodos de la clase

    // El método main, que indica donde empieza la ejecución
    public static void main(String[] args) {
        // Declaración de las variables del método

        // Sentencias de ejecución del método

    }
}
```

En el caso de ser una clase distinta de la clase principal, el método `main()` no suele aparecer.

Se han puesto como comentarios (que empiezan por dos barras) los elementos que suelen componer una clase, que son los siguientes:

- **Atributos de la clase.** Son las variables que definen el estado de los objetos. Se tratarán con detalle en el Capítulo 2. Puede pensar en ellos como variables definidas en el bloque de la clase.
- **Declaración de métodos.** Son fragmentos de código que hacen una determinada función. Se tratarán con detalle en el Capítulo 2.
- **El método principal main().** Ya se ha visto en el primer ejemplo. Dentro de este método se escribe lo que se desea que ejecute el programa, ya que siempre se empieza a ejecutar este método.
- **Declaración de variables.** Se verán en las siguientes secciones.
- **Sentencias.** Es la parte que se ejecuta en el programa. Ya se ha visto en el primer ejemplo un programa con una sentencia para escribir un texto en la pantalla. Las sentencias se ejecutan siempre en el orden en el que se escriben y, siempre, una detrás de otra, hasta que se acaba el método `main()`.

⇨ Sobre programas básicos de escritura de datos consulte los Ejercicios 1.1 a 1.3.

1.2 IDENTIFICADORES

Un identificador es un nombre. Los nombres permiten identificar los elementos que se están manejando en un programa. Existen reglas estrictas sobre cómo se pueden escribir los nombres de las clases, las variables, métodos, etc. Cualquiera de estos nombres debe ser un *identificador*. Un identificador debe empezar con una letra y debe seguir una sucesión de letras y dígitos.

Una letra es cualquier carácter que se considera una letra en Java. Como en Java se utiliza Unicode para los caracteres, un identificador se puede escribir con caracteres hebreos, cirílicos, armenios, katakana, etc. Para formar un identificador también se consideran letras los caracteres subrayado '_' y dólar '$', aunque el carácter '$' prácticamente no se suele utilizar, salvo en algún tipo de nombrado automático. Se considera un dígito a cualquier carácter entre los caracteres '0' a '9'. De esta forma son válidos los siguientes identificadores:

```
ejemplo, EjemploDeIdentificador, εφεμπλο, otroEjemplo12, uno2tres4oMás, AñoMás,
_víó_LôQüëö_Nô_3303459345
```

Aunque de acuerdo con la regla anterior serían válidos, sin embargo, no lo son los identificadores que coincidan con una palabra reservada del lenguaje (tenga cuidado porque `final` es una palabra reservada).

Cuando elija un identificador, siga las siguientes recomendaciones:

- Debe utilizar nombres que sean significativos, de forma que cuando lo vea escrito sepa para qué sirve y de qué tipo va a ser sin necesidad de acudir a su declaración.
- Los nombres de variables y métodos empiezan con minúscula. Si se trata de un nombre compuesto, cada palabra empieza con mayúscula. No se utiliza el carácter subrayado para separar unas de otras. Ejemplos de nombres de variables o métodos: `n`, `númeroElementos`, `ponValor`, `escribeTítulo`.
- Los nombres de clases empiezan con mayúscula. Si se trata de un nombre compuesto, cada palabra empieza con mayúscula. No se utiliza el carácter subrayado para separar unas de otras. Ejemplos de nombres de clases: `VolumenCilindro`, `Alumno`, `ProgramaDePrueba`.
- Los nombres de constantes se escriben en mayúsculas. Si el nombre es un nombre compuesto utilice el carácter subrayado para separar unos de otros. Ejemplos de nombres de constantes: `PI`, `TAMAÑO_MÁXIMO`.

⇨ Sobre identificadores consulte los Ejercicios 1.4 y 1.5.

1.3 TIPOS, VARIABLES Y VALORES

Un programa maneja valores, maneja datos de forma apropiada para cambiarlos, hacer cálculos, presentarlos, solicitarlos al usuario, escribirlos en un disco, enviarlos por una red, etc.

Para poder manejar los valores en un programa se guardan en variables. Una variable guarda un único valor. Una variable queda determinada por:

- Un nombre. Este nombre debe de ser como se ha indicado en la sección de identificadores.
- Un tipo. Permite conocer qué valores se pueden guardar en dicha variable.
- Un rango de valores que puede admitir. Viene determinado por el tipo de la variable.

Por ejemplo, si se tiene una variable de nombre `númeroElementos`, donde el tipo de valores que se pueden guardar son números enteros, `númeroElementos` puede contener el número 34, o el número –234 (aunque no tenga sentido contar elementos negativos, la variable podría contener ese valor). Pero nunca puede contener el valor 3.45 ni un texto como "ejemplo de texto", ni un valor mayor que el admitido por la variable, como por ejemplo 239849695287398274832749. Para declarar una variable de un tipo, se indica de la siguiente forma:

```
double radio;
```

Con ello se declara que va a existir en el programa una variable con el nombre radio y que esa variable va a guardar un valor del tipo `double`. Una vez declarada se puede utilizar en cualquier lugar del programa poniendo su nombre. Siempre que se utilice el nombre de una variable es como si pusiese el valor que tiene.

Si se quiere guardar un valor en una variable, se utiliza el operador de asignación de valor de la siguiente forma:

```
radio = 23.4;
```

Donde se indica el nombre de la variable, el carácter igual (=), y cualquier expresión o cálculo que se desee. El símbolo igual significa lo siguiente: haz el cálculo de la expresión que se encuentra a la derecha del igual y, después, guarda el valor calculado en la variable que hay a la izquierda. Si después se ejecuta:

```
radio = 44.56;
```

la variable radio guarda el valor 44.56. El valor anterior se pierde en esa variable.

Se puede declarar una constante poniendo el modificador final:

```
final double PI = 3.1415926536;
```

El valor de una constante no se puede modificar en el programa, por eso hay que darle un valor a la vez que se declara.

Los tipos primitivos que se pueden utilizar para declarar variables, y los intervalos de valores de cada uno de ellos, se pueden ver en la Tabla 1.1.

Tabla 1.1. Tipos primitivos en Java 2.

Tipo	Descripción	Valor min./max.
`byte`	Entero con signo	-128 a 127
`short`	Entero con signo	-32768 a 32767
`int`	Entero con signo	-2147483648 a 2147483647
`long`	Entero con signo	-9221170368547758O8 a 922117036854775807
`float`	Real de simple precisión	±3.40282347e+38 a ±1.40239846e–45
`double`	Real de doble precisión	±1.7976931348623157 0e+308 a ±4.94065645841246544e–324
`char`	Caracteres Unicode	\u0000 a \uFFFF
`boolean`	Verdadero o falso	true o false

Los tipos primitivos se pueden clasificar en:

- **Números enteros.** Permiten representar números enteros positivos y negativos con distintos intervalos de valores.
- **Números reales.** Permiten guardar valores con decimales con distinta precisión.
- **Caracteres.** Existe un tipo carácter (`char`) que permite representar cualquier carácter Unicode.
- **Booleano.** Es un tipo que indica un valor lógico. Sólo tiene dos valores, verdadero (`true`) y falso (`false`).

Otro tipo muy utilizado es para los textos o cadenas de caracteres. En realidad se trata de objetos, que se verán con detenimiento en los Capítulos 2 y 3, pero se pueden utilizar de forma sencilla como si fueran variables de tipos primitivos.

```
String texto;
texto = "En un lugar de la mancha de cuyo nombre...";
```

⇨ Sobre el uso básico de variables consulte los Ejercicios 1.11 a 1.13.

Literales

Números enteros

Los números enteros se pueden escribir de tres formas:

- **En decimal**: 21. Es la forma habitual.
- **En octal**: 025. En octal un número siempre empieza por cero, seguido de dígitos octales (del 0 al 7).
- **En hexadecimal**: 0x15. En hexadecimal un número siempre empieza por 0x seguido de dígitos hexadecimales: del 0 al 9, de la 'a' a la 'f' y de la 'A' a la 'F'.

Si se escribe un valor del tipo `long` se le debe añadir detrás el carácter 'l' o 'L' (una letra ele minúscula o mayúscula).

Números reales
Para escribir valores reales en Java, se puede hacer de las siguientes formas: 1e2, 2., .54, 0.45, 3.14, 56.34E-45. Es decir, un número real en Java siempre tiene que tener un punto decimal o un exponente indicado por la letra e minúscula o la letra E mayúscula.

Si no se indica nada se supone que pertenecen al tipo `double`. Si se desea que se interpreten como del tipo float se debe añadir un carácter 'f' o 'F' detrás del valor de la siguiente forma: 1e2f, 2.f, .54f, 0.45f, 3.14f. Se puede añadir el carácter 'd' o el carácter 'D' para indicar explícitamente que el valor es del tipo `double`, de la siguiente forma: 4.56d, 78.34e-4d.

Booleanos
Los valores del tipo `boolean` sólo pueden ser dos, `true` y `false`, y se escriben siempre en minúsculas.

Caracteres
Los valores del tipo carácter representan un carácter Unicode. Un carácter siempre se escribe entre comillas simples. Un valor carácter se escribe como 'a', 'Z', 'Ñ', ';', 'π', etc., o por su código de la tabla Unicode, en octal o en hexadecimal. Por ejemplo: '\u00A3', en hexadecimal o '\102' en octal.

Existen algunos caracteres especiales como los que se indican en la Tabla 1.2.

Tabla 1.2. Caracteres especiales en Java.

Carácter	Significado
\b	Retroceso
\t	Tabulador
\n	Salto de línea
\r	Cambio de línea
\"	Carácter comillas dobles
\'	Carácter comillas simples
\\	Carácter barra hacia atrás

1.4 EXPRESIONES

Sobre cada uno de los tipos de valores se pueden utilizar un conjunto de operadores para formar expresiones o cálculos.

Números enteros
Al realizar una operación entre dos números enteros el resultado **siempre** es un entero.

- Unarias: poner un signo más o un signo menos delante. Por ejemplo: +44, -56.
- Multiplicativas: * multiplica dos valores, / divide el primer valor entre el segundo, y % calcula el resto de la división entera. Ejemplos: 4 * 5, 8 / 2, 5 % 2.
- Aditivas (+, –): La suma y la resta de la forma usual. Ejemplo: 4 + 5
- Incremento y decremento (++, – –): Incrementa el valor en uno y decrementa el valor en uno de una variable. Los operadores de incremento y decremento se pueden poner antes o después de la variable que se desea incrementar o decrementar.
- Relación (>, >=, <, <=): Permiten comparar valores. El resultado de una operación con los operadores de relación es un valor `boolean` indicando si es cierta o falsa la relación.
- Operadores de igualdad (==, !=): Comparan si dos valores son iguales o son distintos. El resultado es un valor del tipo `boolean`, indicando si es cierta o no la igualdad o desigualdad.
- Operadores de asignación (=, +=, -=, *=, /=, %=): El primero es el operador de asignación ya descrito y el resto son operadores que permiten simplificar la escritura de expresiones muy comunes.

Números reales

Con los números reales se pueden realizar las mismas operaciones que con números enteros. En el caso de las operaciones unarias, aditivas o multiplicativas el resultado de la operación con números reales es un número real. También se pueden utilizar los operadores de relación e igualdad cuyo resultado es un valor del tipo `boolean`.

Booleanos

Los operadores sobre booleanos son los siguientes:

- Negación (!): Devuelve `true` si el operando vale `false` y viceversa.
- Y lógico (&&): Devuelve `false` si el primer operando vale `false`. En otro caso, devuelve lo que valga el segundo operando. También existe la versión con un solo ampersand (&) en cuyo caso siempre se evalúan los dos operandos.
- O lógico (||): Devuelve `true` si el primer operando vale `true`. En otro caso, devuelve lo que valga el segundo operando. También existe la versión con una sola barra vertical (|) en cuyo caso siempre se evalúan los dos operandos.

Precedencia de operadores

Toda expresión se evalúa a un valor de una forma estricta. El cómo se evalúa una expresión depende del orden de prioridad de los operadores que contenga dicha expresión. De forma simplificada el orden de prioridad y la forma de evaluación es:

1. Operadores unarios.
2. Operadores multiplicativos, de izquierda a derecha.
3. Operadores aditivos, de izquierda a derecha.
4. Operadores de relación.
5. Operadores de asignación.

Teniendo en cuenta el orden de evaluación, la siguiente sentencia:

```
a = -3 + 5 + 2 * 4 - 6 / 4 * 3 - 5 % 2;
   ①⑥  ⑦  ②  ⑧  ③  ④  ⑨  ⑤
```

evalúa la expresión de acuerdo al orden indicado debajo de la misma. De esta forma primero se aplica el operador unario a 3 para obtener el valor –3. A continuación se van evaluando los operadores multiplicativos de izquierda a derecha, 2 * 4 se evalúa a 8, 6 / 4 se evalúa a 1 que multiplicado por 3 se evalúa a 3 y 5 % 2 se evalúa a 1. En este momento la expresión queda:

```
a = -3 + 5 + 8 - 3 - 1;
```

Por último, se evalúan los operadores aditivos de izquierda a derecha, siendo el resultado final de 6. Este es el valor que se guarda en la variable `a`.

Si se desea que la evaluación se realice en un orden específico, se deben utilizar paréntesis. En una expresión siempre se empieza a evaluar por los paréntesis más internos.

⇨ Sobre la evaluación de expresiones aritméticas consulte los Ejercicios 1.9 y 1.10.

Expresiones aritmético-lógicas

Una expresión aritmético-lógica es una expresión que devuelve un valor booleano donde se utilizan operadores aritméticos y operadores relacionales y de igualdad. Una expresión aritmético-lógica podría ser la que sigue y su valor es `true`, pues es cierto que 8 es menor que 10:

```
(3 + 5) < (5 * 2)
```

En una expresión aritmético-lógica se pueden combinar varias expresiones sencillas de las anteriores mediante los operadores lógicos. La precedencia de los operadores booleanos es menor que la de los operadores relacionales, por lo que primero se evalúan las desigualdades y después los operadores booleanos. El orden de prioridad entre los operadores booleanos es: la negación, después el Y lógico y, por último, el O lógico. La prioridad de los operadores de asignación es la menor de todas. Por tanto, en la expresión:

```
3+5 < 5*2 || 3 > 8 && 7 > 6
```

se evalúa primero las expresiones aritméticas y después las relacionales, quedando la expresión

```
true || false && true
```

En realidad se evalúa en primer lugar la primera expresión a `true`, para, como el operador es ||, evaluar la expresión completa a `true`.

➪ Sobre la evaluación de expresiones aritmético-lógicas consulte los Ejercicios 1.14 y 1.15.

1.5 CONVERSIONES DE TIPO

En muchas ocasiones resulta necesario realizar algunas conversiones de tipos, de forma que el resultado sea del tipo esperado. Para convertir valores entre tipos existen dos formas:

- Conversión automática de tipos. Cuando el tipo al que se asigna un valor es "mayor", la conversión se realiza automáticamente. Así un valor de tipo `double` se puede multiplicar por otro del tipo `int`. Antes de hacer la multiplicación se convierte el valor `int` a `double` y luego se hace la multiplicación entre reales.
- Conversión explícita. Se pone delante del valor a convertir, entre paréntesis, el tipo al que se desea convertir. Como esta conversión suele implicar una pérdida de valor, hay que tener mucho cuidado. Por ejemplo `(int)34.45`, hace una conversión a un tipo `int`, trunca el valor y se pierden los decimales.

En la conversión de tipos existe un tipo especial de expresiones que involucra a los valores del tipo `char`. Un tipo `char` siempre se puede utilizar en una expresión junto con números enteros como si fuese un número entero más:

```
char c = 'A';
int n;
n = c + 2;
```

1.6 ENUMERADOS

Los enumerados son conjuntos de valores constantes para los que no existe un tipo predefinido. Por ejemplo, no existe ningún tipo predefinido para representar los días de la semana, las estaciones del año, los meses del año, los turnos de clases, etc.

Para definir un tipo enumerado con sus valores se haría de la siguiente forma:

```
enum DíaSemana {LUNES, MARTES, MIÉRCOLES, JUEVES, VIERNES, SABADO, DOMINGO}
enum TurnoDeClase {MAÑANA, TARDE}
enum TipoDeClase {TEORIA, LABORATORIO, SEMINARIO, CHARLA, EXPERIMENTO}
```

En el siguiente ejemplo se manejan los valores de los días de la semana.

```
public class Dias {
    public enum DíaSemana {LUNES, MARTES, MIÉRCOLES, JUEVES, VIERNES, SABADO, DOMINGO}

    public static void main(String[] args) {
        DiaSemana hoy = DíaSemana.JUEVES;
        DiaSemana último = DíaSemana.DOMINGO;

        System.out.println("Hoy es " + hoy);
        System.out.println("El ultimo día es " + último);
    }
}
```

⇨ Sobre el uso de enumerados consulte los Ejercicios 1.18 a 1.20.

1.7 PETICION DE VALORES PRIMITIVOS AL USUARIO

Como ya habrá imaginado lo habitual en un programa es que solicite datos al usuario para realizar los cálculos del programa. En el siguiente ejemplo se presenta un programa para el cálculo del volumen de un cilindro en el que se piden al usuario los datos del radio y la altura.

```
/**
 * Programa en Java que pide al usuario los datos
 * del radio y la altura de un cilindro y calcula su volumen
 */
import java.util.Scanner;
public class PedirDatos {
    public static void main(String[] args) {
        // El valor del numero pi
        final double PI = 3.1415926536;
        double radio;
        double altura;

        Scanner teclado = new Scanner(System.in);

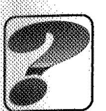
        System.out.println("Introduzca los datos del cilindro:");
        System.out.print("Radio: ");
        radio = teclado.nextDouble();
        System.out.print("Altura: ");
        altura = teclado.nextDouble();
        System.out.print("El área del cilindro es: ");
        System.out.println(PI * radio * radio * altura);
    }
}
```

En la primera línea marcada se declara un objeto de la clase Scanner. Ya verá en el Capítulo 2 qué es un objeto y cómo se utiliza. De momento, piense que la variable teclado va a ser un objeto que nos va a permitir leer los datos que se escriben por el teclado. En la segunda línea marcada, cuando el programa la ejecuta se queda esperando a que el usuario escriba algo en el teclado y pulse la tecla de retorno. En ese momento convierte lo leído en un valor del tipo double y lo guarda en la variable radio. Igual ocurre con la tercera línea marcada para leer un valor para la altura. Si lo que desea es leer otro tipo de datos hágalo como se indica en la Tabla 1.3.

Tabla 1.3. Métodos para la lectura de distintos tipos de datos.

Tipo	Método a invocar
`byte`	`teclado.nextByte();`
`short`	`teclado.nextShort();`
`int`	`teclado.nextInt();`
`long`	`teclado.nextLong();`
`float`	`teclado.nextFloat();`
`double`	`teclado.nextDouble();`
`boolean`	`teclado.nextBoolean();`

⇨ Sobre la petición de datos al usuario consulte los Ejercicios 1.21 a 1.24.



Problemas resueltos

PROGRAMAS DE ESCRITURA DE DATOS

Ejercicio 1.1:

Escriba un programa que escriba en la pantalla su nombre completo en una línea y en la línea siguiente su fecha de nacimiento.

Planteamiento: Para escribir este programa se va a utilizar el esquema básico donde la clase tiene el nombre `MisDatosPersonales` (fíjese en las mayúsculas y minúsculas). Dentro del bloque del `main()` se utilizarán dos sentencias para escribir en pantalla, una para el nombre y otra para la fecha de nacimiento. Ambos datos se escribirán como textos entre comillas dobles.

Solución:

```
public class MisDatosPersonales {
      public static void main(String[] args) {
          System.out.println("José Juan Juárez Juárez");
          System.out.println("12/12/1977");
     }
}
```

Para escribir un texto se pone entre comillas dobles.

Comentarios: Se utiliza `System.out.println()` para escribir un texto y pasar a la línea siguiente.

Ejercicio 1.2:

Escriba un programa que imprima en la pantalla su nombre completo en una línea y en la línea siguiente su fecha de nacimiento. Para ello escriba una sentencia para escribir el nombre, otra sentencia para escribir su primer apellido y otra para escribir su segundo apellido.

Planteamiento: Se va a escribir un programa de nombre `MisDatosPersonales2` que escriba el nombre, que se hará sin cambiar de línea, a continuación escriba el primer apellido sin cambiar de línea y, después, el segundo apellido y cambie de línea. Para terminar escribe la fecha de nacimiento.

Solución:

```
public class MisDatosPersonales2 {
```

```
        public static void main(String[] args) {
            System.out.print("José Juan "); ←
            System.out.print("Juárez ");
            System.out.println("Juárez");
            System.out.println("12/12/1977");
        }
}
```

Hay que escribir un espacio para que el apellido aparezca separado.

Ejercicio 1.3:

Escriba un programa que escriba en pantalla las notas de la asignatura de "Programación". En la primera línea se escribirá el nombre de la asignatura. En las siguientes líneas se escribirán las notas de los dos parciales realizados poniendo la nota de cada uno en líneas distintas. En la última línea escribirá la nota final de la asignatura. Escriba lo que sea texto como un texto entre comillas dobles y lo que sea un número como un número.

Planteamiento: Se va a escribir un programa de nombre `MisNotas` que escriba los textos tal y como se ha hecho en los ejercicios anteriores. Para escribir las notas se utilizará la misma sentencia pero poniendo entre paréntesis el número de la nota. Hay que tener cuidado porque en Java los números con decimales utilizan el punto decimal.

Solución:

```
public class Notas  {
        public static void main(String[] args) {
            System.out.println("Notas de Programación");
            System.out.print("Nota del primer parcial: ");
            System.out.println(4.5); ←
            System.out.print("Nota del segundo parcial: ");
            System.out.println(7.5);
            System.out.print("Nota final: ");
            System.out.println(6);
        }
}
```

Para los decimales hay que utilizar el punto decimal.

IDENTIFICADORES

Ejercicio 1.4:

Dados los siguientes identificadores que se van a utilizar en un programa escrito en Java, diga cuáles de ellos son correctos y cuáles no. Justifique su respuesta.

a) *mi carta*
b) *unacarta*
c) *mis2escritos*
d) *4cientos*
e) *es_un_mensaje*
f) *no_vale nada*
g) *_____ejemplo______*
h) *mi-programa*
i) *¿cuantos?*
j) *el%Descontado*
k) *a150PORHORA*
l) *TengoMUCHOS$$$*

m) *LOS400GOLPES*
n) *quieroUNAsolución*

Planteamiento: Hay que tener en cuenta que un identificador sólo puede estar formado por letras y dígitos y que debe comenzar exclusivamente por una letra.
En Java también se consideran letras los caracteres subrayado y $, aunque este último se utiliza sólo de forma especial.

Solución:

a) No es correcto, pues tiene un espacio en medio.
b) Es correcto, pues está formado sólo por letras.
c) Es correcto, pues está formado por letras y tiene un dígito en medio.
d) No es correcto, pues comienza por un dígito y eso no es válido.
e) Es correcto, pues está formado sólo por letras, ya que el carácter subrayado se considera también como una letra.
f) No es correcto, pues tiene un espacio en medio.
g) Es correcto, pues el carácter subrayado se considera una letra. Por tanto, el identificador está formado sólo por letras.
h) No es correcto, pues el carácter guión no se considera una letra ni un dígito.
i) No es correcto, pues los caracteres de abre interrogación y cierra interrogación no se consideran letras ni dígitos.
j) No es correcto, pues el carácter tanto por ciento no se considera una letra ni un dígito.
k) Es correcto, pues está formado por letras y tiene dígitos en medio.
l) Es correcto, pues el símbolo $ se considera también una letra.
m) Es correcto, pues está formado por letras y tiene dígitos en medio.
n) Es correcto, pues está formado sólo por letras.

Ejercicio 1.5:

Dados los siguientes identificadores que se van a utilizar en un programa escrito en Java, diga cuáles de ellos son correctos y cuáles no. Justifique su respuesta.

a) *descarta2*
b) *cuántosQuerrás*
c) *Carr3Mesas*
d) *çaVaBienAvec$$*
e) *Êgresa*
f) *österreich*
g) *Nosÿevan*
h) *EsåCîìÄÅÉæÆôöòûùÿÖÜø£Ø*

Solución:

a) Es correcto, pues está formado sólo por letras y acaba en un dígito.
b) Es correcto, pues está formado sólo por letras.
c) Es correcto, pues está formado sólo por letras y un dígito en medio.
d) Es correcto, pues está formado sólo por letras, ya que la c con cedilla es una letra y el carácter $ se considera una letra.
e) Es correcto, pues está formado sólo por letras, ya que la E tal como está acentuada es una letra.
f) Es correcto, pues está formado sólo por letras, ya que la o con diéresis es una letra.
g) Es correcto, pues está formado sólo por letras, ya que la y con diéresis es una letra.
h) Es correcto, pues está formado sólo por letras.

VARIABLES Y EXPRESIONES ARITMÉTICAS

Ejercicio 1.6:

Escriba un programa que escriba en la pantalla cuánto le costará comprar unas deportivas cuyo precio de catálogo es de 85,00 €, si sabe que puede conseguir una rebaja del 15%.

Planteamiento: Hay que realizar el cálculo de 85,00€ menos el 15%, para ello se utiliza la fórmula (85*(1-0,15)). Hay que tener cuidado, pues en Java los números con decimales utilizan el punto decimal. Se llamará al programa `CompraZapatillas`.

Solución:

```
public class CompraZapatillas {
    public static void main(String[] args) {
          System.out.print(85.00 * (1 - 0.15));
    }
}
```

Se utiliza el punto decimal para los números reales.

Comentario: Al ejecutar el programa en la pantalla aparece `72.25`, cuyo valor también se puede calcular como (85 * 0.85).

Ejercicio 1.7:

Escriba un programa que escriba en la pantalla cuánto le dará su banco después de seis meses si pone 2000€ en una cuenta a plazo fijo al 2,75% anual. Recuerde que al pagarle los intereses el banco le retendrá el 18% para hacienda.

Planteamiento: Hay que realizar el cálculo de 2000€ al 2,75%, es decir, los intereses son 2000*2,75/100/2, ya que seis meses es ½ del año. De esos intereses le retienen el 18%, por lo que realmente el banco le pagará 2000*2,75/100/2 * (1-0,18).

Solución:

```
public class CalculoIntereses {
    public static void main(String[] args) {
          System.out.print(2000*2.75/100/2 * (1-0.18));
    }
}
```

Se utiliza el punto decimal para los números reales.

Comentario: Al ejecutar el programa en la pantalla aparece `22.55`.

Aviso: En la expresión se han mezclado números reales y números enteros. Por tanto, se está haciendo una conversión automática de tipos, convirtiendo todos los valores a reales y haciendo las operaciones entre números reales.

Ejercicio 1.8:

Escriba un programa que escriba en la pantalla cuánto le dará su banco después de seis meses si pone 2000€ en una cuenta a plazo fijo al 2,75% anual. Recuerde que al pagarle los intereses el banco le retendrá el 18% para hacienda. Escriba los mensajes apropiados para entender todos los cálculos.

Planteamiento: Hay que realizar el cálculo de 2000€ al 2,75%, es decir, los intereses son 2000*2,75/100/2, ya que seis meses es ½ del año. De esos intereses le retienen el 18%, por lo que realmente el banco le pagará 2000*2,75/100/2 * (1-0,18).

Solución:

```
public class CalculoIntereses {
```

```
    public static void main(String[] args) {
        System.out.println("Cálculo de intereses.");
        System.out.println("Dinero ingresado: 2000€.");
        System.out.println("Interés anual: 2,75%");
        System.out.println("Intereses a los seis meses: " + (2000*2.75/100/2));
        System.out.println("Retenciones realizadas: " + (2000*2.75/100/2 * 18/100));
        System.out.println("Intereses cobrados: " + (2000*2.75/100/2 * (1 - 0.18)));
    }
}
```

Se han puesto los cálculos entre paréntesis por claridad.

Aviso: Hay que tener cuidado al realizar cálculos e imprimirlos. Si se hubiese realizado una suma en lugar de un producto y se hubiese escrito:

```
System.out.println("La suma de 2 + 2 vale: " + 2+2);
```

se hubiese impreso en la pantalla:

```
La suma de 2 + 2 vale: 22
```

Es decir, los operadores + se evalúan de izquierda a derecha, por lo que se concatena el texto con el primer 2, y el resultado es un texto. A continuación se concatena el texto con el segundo 2, pues hay otra operación +, dando como resultado La suma de 2 + 2 vale: 22. Sin embargo, si se escribe:

```
System.out.println("El producto de 2 * 2 vale: " + 2 * 2);
```

como el operador * tiene mayor precedencia que el operador +, en primer lugar se evalúa el operador *, por lo que 2*2 se evalúa a 4. A continuación, se evalúa la concatenación del texto con el 4, escribiendo, por tanto, en pantalla:

```
El producto de 2 + 2 vale: 4
```

Para escribir correctamente la suma de 2 + 2 hay que escribir la expresión aritmética entre paréntesis:

```
System.out.println("La suma de 2 + 2 vale: " + (2 + 2));
```

Ejercicio 1.9:

Dadas las siguientes expresiones aritméticas, calcule cuál es el resultado de evaluarlas.

a) $25 + 20 - 15$
b) $20 * 10 + 15 * 10$
c) $20 * 10 / 2 - 20 / 5 * 3$
d) $15 / 10 * 2 + 3 / 4 * 8$

Planteamiento: Para cada una de las expresiones anteriores hay que tener en cuenta la precedencia de los operadores. Como sólo constan de operadores aditivos y multiplicativos se harán primero los multiplicativos de izquierda a derecha y luego los aditivos de izquierda a derecha. Hay que tener en cuenta también que como todos los números son números enteros el resultado de todas las operaciones serán números enteros.

Solución:

a) $25 + 20 - 15$
Se realizan las operaciones de izquierda a derecha, pues son todas aditivas:
$25 + 20 - 15 \equiv 45 - 15 \equiv 30$

b) 20 * 10 + 15 * 10
Se realizan primero las operaciones multiplicativas de izquierda a derecha:
20 * 10 + 15 * 10 ≡ 200 + 15 * 10 ≡ 200 + 150
Después se realiza la operación de suma
200 + 150 ≡ 350

c) 20 * 10 / 2 – 20 / 5 * 3
Se realizan primero las operaciones multiplicativas de izquierda a derecha:
20 * 10 / 2 – 20 / 5 * 3 ≡ 200 / 2 – 20 / 5 * 3 ≡ 100 – 20 / 5 * 3 ≡ 100 – 4 * 3 ≡ 100 – 12
Después se realiza la operación de sustracción
100 – 12 ≡ 88

d) 15 / 10 * 2 + 3 / 4 * 8
Se realizan primero las operaciones multiplicativas de izquierda a derecha:
15 / 10 * 2 + 3 / 4 * 8 ≡ 1 * 2 + 3 / 4 * 8 ≡ 2 + 3 / 4 * 8 ≡ 2 + 0 * 8 ≡ 2 + 0
Después se realiza la operación de suma
2 + 0 ≡ 2

Comentarios: Hay que recordar que las operaciones entre valores enteros siempre son enteros. Por ello, en el apartado d) 15/10 se evalúa a 1 y, de la misma forma, 3/4 se evalúa a 0.

Ejercicio 1.10:

Dadas las siguientes expresiones aritméticas, calcule cuál es el resultado de evaluarlas. Suponga que las variables a y b que aparecen son del tipo `int` *y a tiene el valor 2 y b tiene el valor 4.*

*a) –a + 5 % b – a * a*
*b) 5 + 3 % 7 * b * a – b % a*
*c) (a + 1) * (b + 1) – b / a*

Planteamiento: En cada una de las expresiones anteriores, cuando sea necesario el valor sustituye la variable por su valor. En el cálculo hay que tener en cuenta la precedencia de los operadores. Hay que tener en cuenta también que como todos los números son números enteros el resultado de todas las operaciones serán números enteros.

Solución:

a) –a + 5 % b – a * a
Antes de operar se va realizando la sustitución de valores
Primero se evalúa la operación unaria del signo – delante de la variable a. El – 2 se evalúa al valor -2 (se representa por un signo menos más pequeño)
– 2 + 5 % 4 – 2 * 2 ≡-2 + 5 % b – a * a
Después se evalúan las operaciones multiplicativas de izquierda a derecha
-2 + 5 % b – a * a ≡-2 + 5 % 4 – a * a ≡-2 + 1 – a * a ≡-2 + 1 – 2 * 2 ≡-2 + 1 – 4
Por último se evalúan las operaciones aditivas de izquierda a derecha
-2 + 1 – 4 ≡-1 – 4 ≡-5

b) 5 + 3 % 7 * b * a – b % a
Se evalúan las operaciones multiplicativas de izquierda a derecha
5 + 3 % 7 * b * a – b % a ≡5 + 3 * b * a – b % a ≡5 + 3 * 4 * a – b % a ≡5 + 12 * a – b % a ≡5 + 12 * 2 – b % a ≡5 + 24 – b % a ≡5 + 24 – 4 % 2 ≡5 + 24 – 0
Por último se evalúan las operaciones aditivas de izquierda a derecha
5 + 24 – 0 ≡29 – 0 ≡29

c) (a + 1) * (b + 1) – b / a
En primer lugar se evalúan las expresiones entre paréntesis
(2 + 1) * (b + 1) – b / a ≡ 3 * (b + 1) – b / a ≡ 3 * (4 + 1) – b / a ≡ 3 * 5 – b / a

Después se evalúan las operaciones multiplicativas de izquierda a derecha
3 * 5 – b / a ≡ 15 – b / a≡ 15 – 4 / 2 ≡ 15 – 2
Por último se evalúa la operación de sustracción
15 – 2 ≡ 13

Comentarios: Cuando en una expresión hay expresiones entre paréntesis, en primer lugar se evalúan los paréntesis más internos.

Ejercicio 1.11:

Escriba un programa que defina dos variables enteras para describir las longitudes de los lados de un rectángulo. El programa debe calcular y escribir en la pantalla las longitudes de los lados, el perímetro y el área del rectángulo. (Suponga que el rectángulo mide 15cm de alto y 25cm de ancho.)

Planteamiento: Se necesita declarar dos variables para guardar los valores de los lados. Como se dice que los lados son números enteros, las variables se declaran del tipo `int`. Para realizar los cálculos se van a declarar dos variables más, una para el cálculo del perímetro y otra para el cálculo del área. Finalmente, se imprimirán por pantalla todos los datos y resultados de los cálculos.

Solución:
```
public class Rectangulo {
    public static void main(String[] args) {
        int alto = 15;
        int ancho = 25;

        int perímetro = 2 * alto + 2 * ancho;
        int área = ancho * alto;

        System.out.print("El rectángulo mide " + alto + " de alto ");
        System.out.println("y " + ancho + " de ancho.");
        System.out.println("El perímetro del rectángulo es: " + perímetro);
        System.out.println("El área del rectángulo es: " + área);
    }
}
```

Concatenación de textos y valores de forma mezclada.

Comentarios: Fíjese en los nombres elegidos para las variables. Es importante que se entienda bien para qué sirve cada valor. Fíjese también en la estructura del programa: primero la declaración de variables y valores iniciales, un segundo bloque de cálculos a partir de las variables y un tercer bloque de presentación de resultados. Tanto la elección de nombres como la estructura utilizada facilitan seguir el programa y entender lo que hace fácilmente. Imagínese la diferencia si las variables se hubiesen llamado `a`, `b`, `c` y `d`. Sería casi imposible saber para qué sirve el programa.

Ejercicio 1.12:

Escriba un programa para calcular el área y el volumen de un cilindro. Para ello declare una constante que guarde el valor de π. Declare, también, variables para el diámetro y la altura del cilindro. Suponga para el ejemplo que el cilindro tiene un diámetro de 15,5cm y una altura de 42,4cm.

Planteamiento: Se necesita declarar dos variables para guardar los valores del diámetro y la altura del cilindro, que debe ser del tipo `double`. Además, hay que declarar una constante para el valor de π. El cálculo del área y el volumen se realiza con las fórmulas clásicas para ello.

Solución:
```
public class AreaVolumenCilindro {
```

```
    public static void main(String[] args) {
        final double PI = 3.14159265; <-- Se declara PI como una constante.

        double diámetro = 15.5; // en cm.
        double altura   = 42.4; // en cm.

        double radio   = diámetro / 2;
        double área    = 2 * PI * radio * radio + 2 * PI * radio * altura;
        double volumen = PI * radio * radio * altura;

        System.out.print("Para un cilindro de radio " + radio);
        System.out.println(" y altura " + altura);
        System.out.println("El área es: " + área);
        System.out.println("El volumen es: " + volumen);
    }
}
```

Comentario: Para utilizar el número PI podría haberse utilizado el que ya está definido en la biblioteca matemática. Para ello basta sustituir PI en las fórmulas por `Math.PI` (consulte en el Capítulo 2 cómo hacerlo). Quizá el número de decimales que resulta al escribir los valores no sea el más apropiado. Puede dar un formato más apropiado, con sólo dos decimales, utilizando el siguiente fragmento de código:

```
System.out.print("Para un cilindro de radio " + radio);
System.out.println(" y altura " + altura);
System.out.printf("El área es: %.2f\n", área);
System.out.printf("El volumen es: %.2f\n", volumen);
```

Además `printf()` realiza una localización de los datos de forma que los números reales aparecen escritos con coma decimal como se hace habitualmente. El % significa que a continuación aparecerá una definición de formato. La expresión %.2f significa escribe un número real con dos dígitos después de la coma. En el mismo punto del texto donde se escribe esa definición de formato se escribirá el valor de la variable que aparece a continuación. El carácter \n al final del texto en el método `printf()` es el carácter de fin de línea.

Ejercicio 1.13:

Escriba un programa que escriba en la pantalla cuánto le dará su banco después de seis meses si pone 2000€ en una cuenta a plazo fijo al 2,75% anual. Recuerde que al pagarle los intereses el banco le retendrá el 18% para hacienda. Utilice variables para manejar las cantidades y realizar los cálculos. Escriba los mensajes apropiados para entender todos los cálculos.

Planteamiento: Para el cálculo se necesitarán variables para guardar el dinero a invertir (`double`), el tipo de interés anual (`double`) y el número de meses de la inversión (`int`). Como el tipo de retención que se aplica es fijo se declarará como una constante de tipo `double` (pues pudiera ser un número no entero). También se utilizará una constante que guarde el número de meses que tiene el año.

Solución:

```
public class CalculoIntereses {
    public static void main(String[] args) {
        final double RETENCIÓN = 18;
        final int mesesAño = 12;

        double capitalInvertido = 2000; //en euros
```

```
        double interésAnual = 2.75;
        int mesesInversión = 6;

        double interesesObtenidos = capitalInvertido *
              interésAnual / 100 *
              mesesInversión / mesesAño;
        double retención = interesesObtenidos * RETENCIÓN / 100;
        double interesesCobrados = interesesObtenidos - retención;

        System.out.println("Cálculo de intereses.");
        System.out.printf("Dinero ingresado: %.2f€.\n", capitalInvertido);
        System.out.printf("Interés anual: %.2f%%.\n", interésAnual);
        System.out.printf("Intereses a los %d meses: %.2f€.\n",
                          mesesInversión, interesesObtenidos);
        System.out.printf("Retención realizada: %.2f€.\n", retención);
        System.out.printf("Intereses cobrados: %.2f€.\n", interesesCobrados);
    }
}
```

Se hace conversión automática de tipo de int a double.

Se imprimen dos valores en orden de aparición.

Comentario: Para escribir valores monetarios resulta muy apropiado el uso de `printf()` (impresión con formato) pues permite decidir que se desean imprimir sólo dos decimales. El formato `%d` indica escribir un número entero y `%f`, un número real. Si aparece una definición de formato (siempre empieza con %), se sustituye cada una de ellas por las variables que se pongan a continuación separadas por comas en el mismo orden de aparición. Los tipos de los valores deben coincidir con el formato indicado para ellos. Como % indica el principio de definición de formato, si se desea escribir el signo % hay que ponerlo dos veces.

EXPRESIONES ARITMÉTICO-LÓGICAS

Ejercicio 1.14:

Dadas las siguientes expresiones aritmético-lógicas calcule cuál es el resultado de evaluarlas.

a) *25 > 20 && 13 > 5*
b) *10 + 4 < 15 – 3 || 2 * 5 + 1 > 14 – 2 * 2*
c) *4 * 2 <= 8 || 2 * 2 < 5 && 4 > 3 + 1*
d) *10 <= 2 * 5 && 3 < 4 || !(8 > 7) && 3 * 2 <= 4 * 2 – 1*

Planteamiento: Para cada una de las expresiones anteriores hay que tener en cuenta que se trata de expresiones aritméticas junto con operaciones de comparación y operaciones relacionales. De acuerdo con la precedencia de las operaciones habrá que realizar las operaciones en el siguiente orden: operaciones aritméticas, operaciones relacionales y, por último, operaciones booleanas. Las operaciones booleanas tienen la siguiente precedencia: primero la negación, después el y-lógico (&&) y, por último, el o-lógico (||).

Solución:

a) 25 > 20 && 13 > 5
Se realizan las operaciones de relación
25 > 20 && 13 > 5 ≡ true && 13 > 5
Como la parte izquierda del y-lógico (&&) vale true hay que evaluar la parte de la derecha:
true && 13 > 5 ≡ true && true
Ahora se evalúa el &&
true && true ≡ true

b) 10 + 4 < 15 – 3 || 2 * 5 + 1 > 14 – 2 * 2
Se evalúa la parte izquierda del operador o-lógico (||). Para ello se evalúa la relación <, y para hacerlo hay que evaluar las expresiones aritméticas a su izquierda y a su derecha
10 + 4 < 15 – 3 || 2 * 5 + 1 > 14 – 2 * 2 ≡ 14 < 15 – 3 || 2 * 5 + 1 > 14 – 2 * 2 ≡ 14 < 12 || 2 * 5 + 1 > 14 – 2 * 2
Se evalúa el operador <
14 < 12 || 2 * 5 + 1 > 14 – 2 * 2 ≡ false || 2 * 5 + 1 > 14 – 2 * 2
Como la parte izquierda del operador || vale false hay que evaluar su parte derecha
false || 2 * 5 + 1 > 14 – 2 * 2 ≡ false || 10 + 1 > 14 – 2 * 2 ≡ false || 11 > 14 – 2 * 2 ≡ false || 11 > 14 – 8 ≡ false || 11 > 6 ≡ false || true
Por último se evalúa el operador ||
false || true ≡ true

c) 4 * 2 <= 8 || 2 * 2 < 5 && 4 > 3 + 1
Esta expresión se evalúa de izquierda a derecha. Como tiene mayor prioridad el operador && que el operador || se evalúa como si estuviese escrito (e1 || (e2 && e3)), por tanto se evalúa e1 y si su valor es false se evalúa la parte derecha del operador ||, por ello:
4 * 2 <= 8 || 2 * 2 < 5 && 4 > 3 + 1 ≡ 8 <= 8 || 2 * 2 < 5 && 4 > 3 + 1 ≡ true || 2 * 2 < 5 && 4 > 3 + 1
Como la expresión a la izquierda del operador || vale true ya no es necesario evaluar la parte a su derecha
true || 2 * 2 < 5 && 4 > 3 + 1 ≡ true

d) 10 <= 2 * 5 && 3 < 4 || !(8 > 7) && 3 * 2 <= 4 * 2 – 1
Teniendo en cuenta la prioridad de los operadores booleanos la expresión a evaluar tiene la siguiente estructura: ((e1 && e2) || (e3 && e4)). Por ello en primer lugar hay que evaluar la parte izquierda del operador ||, y para ello en primer lugar la parte izquierda del primer operador &&, es decir, e1.
10 <= 2 * 5 && 3 < 4 || !(8 > 7) && 3 * 2 <= 4 * 2 – 1 ≡ 10 <= 10 && 3 < 4 || !(8 > 7) && 3 * 2 <= 4 * 2 – 1 ≡ 10 <= true && 3 < 4 || !(8 > 7) && 3 * 2 <= 4 * 2 – 1
Como la parte izquierda del primer operador && vale true se evalúa su parte derecha
true && 3 < 4 || !(8 > 7) && 3 * 2 <= 4 * 2 – 1 ≡ true && true || !(8 > 7) && 3 * 2 <= 4 * 2 – 1
Por tanto, el resultado de evaluar el primer operador && es true.
true && true || !(8 > 7) && 3 * 2 <= 4 * 2 – 1 ≡ true || !(8 > 7) && 3 * 2 <= 4 * 2 – 1
En este momento como el operador || tiene a su izquierda el valor true, ya no necesita evaluar la parte a su derecha, siendo el valor final de la expresión true
true || !(8 > 7) && 3 * 2 <= 4 * 2 – 1 ≡ true

Ejercicio 1.15:

Dadas las siguientes expresiones aritmético-lógicas calcule cuál es el resultado de evaluarlas. Suponga que las variables a y b que aparecen son del tipo `int` *y a tiene el valor 5 y b tiene el valor 3.*

*a) !(a > b && 2 * a <= b)*
b) b++ > 3 || a + b <= 8 && !(a > b)
c) a++ < 6 && (b += 2) < a
*d) a++ / 2 < b && (a++ / 2 > b || (a * 2 < b * 4))*

Planteamiento: En cada una de las expresiones anteriores se sustituye la variable por su valor cuando se vaya a utilizar. En el cálculo hay que tener en cuenta la precedencia de los operadores. Hay que tener en cuenta también que como todos los números son números enteros el resultado de todas las operaciones serán números enteros.

Solución:

a) !(a > b && 2 * a <= b)
En primer lugar se evalúa la parte entre paréntesis. Para ello se evalúa primero la parte izquierda del operador &&
!(a > b && 2 * a <= b) ≡ !(5 > 3 && 2 * a <= b) ≡ !(true && 2 * a <= b)

Como la parte izquierda del operador && vale true, hay que evaluar su parte derecha
!(true && 2 * a <= b) ≡ !(true && 2 * 5 <= 3) ≡ !(true && 10 <= 3) ≡ !(true && false)
De donde evaluando los operadores booleanos
!(true && false) ≡ !(false) ≡ true

b) b++ > 3 || a + b <= 8 && !(a > b)
Esta expresión se evalúa de izquierda a derecha. Como tiene mayor prioridad el operador && que el operador || se evalúa como si estuviese escrito (e1 || (e2 && e3)), por tanto se evalúa e1 y si su valor es false se evalúa la parte derecha del operador ||, por ello
b++ > 3 || a + b <= 8 && !(a > b) ≡ 3 > 3 || a + b <= 8 && !(a > b) ≡ false || a + b <= 8 && !(a > b)
Además, al evaluar b++, b se incrementa en 1 pasando a valer 4. Como la parte izquierda de la expresión vale false hay que evaluar la parte derecha. Se comienza entonces por la parte izquierda del operador &&
false || a + b <= 8 && !(a > b) ≡ false || 5 + 4 <= 8 && !(a > b) ≡ false || 9 <= 8 && !(a > b) ≡ false || 9 <= 8 && !(a > b) ≡ false || false && !(a > b)
Como la parte izquierda del operador && vale false ya no es necesario evaluar la parte derecha
false || false && !(a > b) ≡ false || false ≡ false

c) a++ < 6 && (b += 2) < a
Esta expresión sólo tiene un operador boolean &&. Por tanto, se evalúa en primer lugar su parte izquierda
a++ < 6 && (b += 2) < a ≡ 5 < 6 && (b += 2) < a ≡ true && (b += 2) < a
y el valor de la variable a se incrementa en 1 a 6. A continuación se evalúa la parte derecha del operador &&. (b += 2) añade a b el valor 2 y devuelve el valor resultado de la asignación
true && (b += 2) < a ≡ true && (5) < a ≡ true && 5 < 6 ≡ true && true ≡ true
El resultado final es true y las variables a y b han quedado con los valores a = 6 y b = 5

d) a++ / 2 < b && (a++ / 2 > b || (a * 2 < b * 4))
Esta expresión consta de un operador booleano (&&) entre dos expresiones, por lo que en primer lugar se evalúa su parte izquierda
a++ / 2 < b && (a++ / 2 > b || (a * 2 < b * 4)) ≡ 5 / 2 < b && (a++ / 2 > b || (a * 2 < b * 4)) ≡ 2 < 3 && (a++ / 2 > b || (a * 2 < b * 4)) ≡ true && (a++ / 2 > b || (a * 2 < b * 4))
Tras obtener el valor de la variable a, esta se ha incrementado en 1, valiendo en este momento 6. Como la parte izquierda del operador && vale true hay que evaluar su parte derecha. Para ello se comienza evaluando la parte izquierda del operador ||
true && (a++ / 2 > b || (a * 2 < b * 4)) ≡ true && (6 / 2 > b || (a * 2 < b * 4)) ≡ true && (3 > 3 || (a * 2 < b * 4)) ≡ true && (false || (a * 2 < b * 4))
Tras obtener el valor de la variable a, esta se incrementa en 1, valiendo en este momento 7. Como la parte izquierda del operador || vale false hay que evaluar su parte derecha
true && (false || (a * 2 < b * 4)) ≡ true && (false || (14 < 12)) ≡ true && (false || (false)) ≡ true && (false) ≡ false
Evaluándose, finalmente, la expresión a false.

Comentario: Cuando en una expresión hay expresiones entre paréntesis, en primer lugar se evalúan los paréntesis más internos.

Ejercicio 1.16:

Dado el siguiente programa, indique qué escribe en pantalla. Justifique su respuesta.

```
public class OperadoresPrePostIncremento {
    public static void main(String[] args) {
        int a=3, b=6, c;
        c = a / b;
        System.out.println("El valor de c es: " + c);
        c = a % b;
        System.out.println("El valor de c es: " + c);
        a++;
```

```
        System.out.println("El valor de a es: " + a);
        ++a;
        System.out.println("El valor de a es: " + a);
        c = ++a + b++;
        System.out.println("El valor de a es: " + a);
        System.out.println("El valor de b es: " + b);
        System.out.println("El valor de c es: " + c);

        c = ++a + ++b;
        System.out.println("El valor de a es: " + a);
        System.out.println("El valor de b es: " + b);
        System.out.println("El valor de c es: " + c);
    }
}
```

Planteamiento: En sentencias donde se utiliza el operador de preincremento y el de postincremento hay que tener cuidado por que si es de postincremento primero se observa el valor de la variable y, una vez se ha utilizado, se incrementa el valor de la misma.

Solución: Se imprime en pantalla lo siguiente:

```
El valor de c es: 0
El valor de c es: 3
El valor de a es: 4
El valor de a es: 5
El valor de a es: 6
El valor de b es: 7
El valor de c es: 12
El valor de a es: 7
El valor de b es: 8
El valor de c es: 15
```

El primer valor de c vale 0 pues se realiza la división entera. El segundo valor que se imprime de c es 3 pues se calcula el resto de la división entera de 3 con 6. La primera vez que se imprime el valor de a vale 4, pues antes de imprimirlo se incrementa su valor. La segunda vez ocurre lo mismo y vale entonces 5, pues se incrementa, esta vez en preincremento antes de imprimir su valor.

Los valores que se imprimen a continuación de a y b son, efectivamente 6 y 7 respectivamente pues ambas se han incrementado en la expresión que hay delante. Sin embargo, el valor que se imprime de c es de 12 pues se realiza la suma de a ya incrementado (6) y b antes de incrementarse (6), ya que se utiliza el operador de postincremento, lo que suma 12. Los últimos valores de a, b y c son, respectivamente, 7, 8 y 15, ya que al imprimir a y b, ya se han incrementado y al guardar el valor en c se hace con los valores ya incrementados, ya que tanto para a como para b se utiliza el operador de preincremento.

Ejercicio 1.17:

Dado el siguiente programa, indique qué escribe en pantalla. Justifique su respuesta.

```
public class ExpresionesConIncrementos {
    public static void main(String[] args) {
        boolean expresión;
        int a = 7;

        expresión = 2 * 5 < 5 * 2 || a + 1 < 10 && ++a % 2 == 0;
```

```
        System.out.println("El valor de la expresión es: " + expresión);
      expresión = 3 < 2 || ++a > 6;
        System.out.println("El valor de la expresión es: " + expresión);

      expresión = a++ < 10 && a % 2 == 0 && a <= 10;
        System.out.println("El valor de la expresión es: " + expresión);

      expresión = a++ < 10 || a % 3 == 2;
        System.out.println("El valor de la expresión es: " + expresión);

      System.out.println("El valor de a es: " + a);
   }
}
```

Planteamiento: Al evaluar las expresiones aritmético-lógicas, como ya se ha visto en los Ejercicios 1.14 y 1.15, la dificultad adicional es que hay que llevar cuenta de cómo se realiza el incremento, bien preincremento o bien postincremento, de la variable a.

Solución: Se imprime en pantalla lo siguiente:

```
El valor de la expresión es: true
El valor de la expresión es: true
El valor de la expresión es: true
El valor de la expresión es: true
El valor de a es: 11
```

En la primera expresión aritmético lógica tanto la parte a la izquierda del operador && como la de su derecha valen true, ya que en la parte de la derecha al hacerse el incremento en preincremento a valdrá 8 y por tanto 8 % 2 vale, ciertamente, 0. En la segunda expresión aritmético lógica el valor de a se incrementa a 9, y este valor sí es mayor que 6 por lo que la expresión vale true. En la tercera expresión aritmético lógica en la primera comparación del valor de a, esta variable vale 9 y, por tanto, sí es menor que 10. Cuando pasa a evaluar a en la siguiente comparación a vale 10, por lo que al hacer 10 % 2, el valor es, efectivamente, 0. En la última comparación también es cierto que 10 <= 10, por lo que la expresión completa vale true.

En la cuarta expresión aritmético lógica en la primera comparación del valor de a, esta variable vale 10 y, por tanto, es falso que sea menor que 10. Sin embargo, cuando se evalúa la parte derecha del operador || a ya se ha incrementado al valor 11, por lo que sí es cierto que 11 % 3 es igual a 2 y, por tanto, la expresión vale true.

Para terminar se imprime el valor de a, que como se puede observar se ha incrementado en cuatro veces, por lo que su valor final es de 11.

ENUMERADOS

Ejercicio 1.18:

Escriba un programa que defina un enumerado para los días de la semana. En el programa defina una variable del enumerado y asígnele el valor del día que corresponde al martes. A continuación, escriba por pantalla dicha variable y escriba el valor del enumerado correspondiente al domingo.

Planteamiento: Los valores del enumerado se escriben, como indica el convenio, todos en mayúsculas. Para declarar una variable para este tipo de dato se utiliza el nombre dado al enumerado. Para asignarle el valor del martes hay que poner delante el nombre del enumerado, un punto y el nombre del valor que corresponde al martes. Para escribir el valor del domingo se pone, de la misma forma, el nombre del enumerado, un punto y el nombre del valor que corresponde al domingo.

Solución:

```
public class Enumerados {

   public enum DíasSemana {LUNES, MARTES, MIERCOLES, JUEVES, VIERNES, SABADO, DOMINGO}

    public static void main(String[] args) {
        DíasSemana unDía = DíasSemana.MARTES;

        System.out.println("El día elegido es: " + unDía);
        System.out.println("El último día de la semana es: " + DíasSemana.DOMINGO);
    }
}
```

Ejercicio 1.19:

Escriba en Java los siguientes tipos enumerados:

a) Los días laborables
b) Los tres primeros meses del año
c) Las calificaciones de un alumno
d) Los colores primarios
e) Las notas musicales
f) Los colores del arco iris
g) Los colores de síntesis de televisión

Planteamiento: Los valores del enumerado se escriben, como indica el convenio, todos en mayúsculas. Para declarar los valores de cada uno de los enumerados se utilizarán los nombres que son de aplicación para cada uno de ellos.

Solución:

a) Los días laborables

```
enum DíasLaborables {LUNES, MARTES, MIÉRCOLES, JUEVES, VIERNES}
```

b) Los tres primeros meses del año

```
enum TresPrimerosMeses {ENERO, FEBRERO, MARZO}
```

c) Las calificaciones de un alumno

```
enum Calificaciones {NO_PRESENTADO, SUSPENSO, APROBADO,
                     NOTABLE, SOBRESALIENTE, MATRÍCULA_DE_HONOR}
```

d) Los colores primarios

```
enum ColoresPrimarios {ROJO, AMARILLO, AZUL}
```

e) Las notas musicales

```
enum NotasMusicales {DO, RE, MI, FA, SOL, LA, SI}
```

h) Los colores del arco iris

```
enum ColoresArcoIris {AMARILLO, ROJO, ANARANJADO, AZUL, VERDE, AÑIL, VIOLETA}
```

i) Los colores de síntesis de la televisión

```
enum ColoresTelevisión {ROJO, VERDE, AZUL}
```

O también en inglés, por ejemplo:

```
enum RGB {RED, GREEN, BLUE}
```

Ejercicio 1.20:

Escriba un enumerado para los tipos de lavado de un túnel de lavado que guarde la información de los tiempos. Los tipos de lavado son básico, normal y super y el tiempo que se tarda en cada uno es de 3, 5 y 8 minutos, respectivamente. Escriba un programa que muestre su funcionamiento.

Planteamiento: Se escribe el enumerado con los valores correspondientes. Hay que declarar un atributo para el tiempo y un método para obtenerlo. Para probarlo en una clase se declara una variable y se usa para ella el método que obtiene el tiempo.

Solución:
```
enum TipoLavado{BASICO(3), NORMAL(5), SUPER(8);
    private int tiempo;
    TipoLavado(int tiempo){
        this.tiempo = tiempo;
    }

    public int tiempo(){
        return tiempo;
    }
}

public class TunelLavado{
    public static void main(String[] args){
        TipoLavado lavadoSuper = TipoLavado.SUPER;

        System.out.println("El lavado " + lavadoSuper + " tarda " + lavadoSuper. tiempo());
    }
}
```

ENTRADA DE DATOS DE USUARIO

Ejercicio 1.21:
Escriba un programa que solicite al usuario una cantidad en segundos y la convierta a días, horas, minutos y segundos.

Planteamiento: Tras solicitar el número de segundos, el programa debe dividir sucesivamente el dato entre 60 para obtener los minutos, entre 60 para obtener las horas y entre 24 para obtener los días.

Solución:
```
import java.util.*;

public class Segundos {
    public static void main(String[] args) {
        int segundosIniciales, segundos, minutos, horas, días;

        Scanner teclado = new Scanner(System.in);

        System.out.print("Introduzca un número de segundos: ");
        segundosIniciales = teclado.nextInt();

        // se calculan los minutos y los segundos que restan
        minutos  = segundosIniciales / 60;
        segundos = segundosIniciales % 60;

        // se calculan las horas y los minutos que restan
        horas   = minutos / 60;
```

```
            minutos = minutos % 60;

            // se calculan los días y las horas que restan
            días  = horas / 24;
            horas = horas % 24;

            System.out.println(segundosIniciales + " segundos son " + días +
                               " días, " + horas + " horas, " +  minutos +
                               " minutos y " + segundos + " segundos.");
        }
    }
```

Ejercicio 1.22:

Escriba un programa que solicite al usuario el tamaño del lado de un triángulo equilátero y calcule su perímetro y su área.

Planteamiento: Tras solicitar el tamaño del lado, que ha de ser un número real, pues es una medida que seguramente no sea un número de metros o de centímetros exactos, se calcula el perímetro como tres veces el lado. Para calcular el área se necesita la base, que es un lado, y la altura, que se obtiene utilizando el teorema de Pitágoras entre un lado y la mitad de la base.

Solución:

```
import java.util.Scanner;

public class TrianguloEquilatero {
    public static void main(String[] args) throws Exception{
      double lado;
      double perímetro, área;

      // Se pide al usuario que introduzca por teclado el valor del lado
      Scanner teclado = new Scanner(System.in);
      System.out.print("Introduzca un valor para el lado: ");
      lado = teclado.nextDouble();

      perímetro = 3 * lado;
      double altura = Math.sqrt(lado * lado - (lado/2.0) * (lado/2.0));
      área = lado * altura / 2;

      System.out.print("El área del triángulo de lado " + lado);
      System.out.println(" es: " + área);

      System.out.print("El perímetro del triángulo de lado " + lado);
      System.out.println(" es: " + perímetro);
    }
}
```

Ejercicio 1.23:

Escriba un programa para calcular el consumo medio de un automóvil. Para ello el programa debe solicitar información sobre las tres últimas veces que se repostó combustible. De la primera solicitará el precio del litro del combustible, el total pagado en llenar el depósito y el número de kilómetros que marcaba el cuenta-kilómetros. De la segunda vez sólo solicitará el precio del litro del combustible y el total pagado en llenar el

depósito, y de la tercera vez, solicitará el valor que indicaba el cuentakilómetros. Con estos datos debe calcular el consumo por cada 100 km y el coste por kilómetro.

Planteamiento: Para calcular el consumo medio hay que tener en cuenta cuántos kilómetros se han hecho con el automóvil y cuántos litros de combustible se han consumido. El total de litros de combustible consumidos es la suma de litros de los dos repostajes realizados, ya que de lo que se reposte en el último no se puede conocer cuántos kilómetros se van a hacer hasta que se consuma el combustible que se ha echado. A partir del precio por litro y el total pagado se puede obtener el número de litros que se han echado en el depósito. Se hará la suma de los litros según se vayan introduciendo los datos. El consumo por cada 100 km se calcula como el consumo total dividido por el número de kilómetros multiplicado por 100. El coste por kilómetro se calcula como lo que se ha pagado en los repostajes dividido por el total de kilómetros realizados.

Solución:
```
import java.util.*;

public class Main {
    public static void main(String[] args) {
      double precioLitro, litros = 0;
      double pagado, coste = 0;
      int kmInicial, kmFinal, kilómetros;

      Scanner teclado = new Scanner(System.in);

      System.out.print("Introduzca el precio por litro del primer repostaje: ");
      precioLitro = teclado.nextDouble();
      System.out.print("Introduzca el coste total del primer repostaje: ");
      pagado = teclado.nextDouble();
      System.out.print("Introduzca el valor del cuentakilómetros en el primer repostaje: ");
      kmInicial = teclado.nextInt();
      litros = pagado/precioLitro;
      coste = pagado;

    System.out.println();
      System.out.print("Introduzca el precio por litro del segundo repostaje: ");
      precioLitro = teclado.nextDouble();
      System.out.print("Introduzca el coste total del segundo repostaje: ");
      pagado = teclado.nextDouble();
      litros += pagado/precioLitro;
      coste += pagado;

      System.out.println();
      System.out.print("Introduzca el valor del cuentakilómetros en el tercer repostaje: ");
      kmFinal = teclado.nextInt();
      kilómetros = kmFinal - kmInicial;

      System.out.println("El consumo medio del automóvil es de " +
                          (litros/kilómetros*100) + " litros por cada 100 Km.");
      System.out.println("El gasto medio es de " + coste/kilómetros + " por kilómetro.");
    }
}
```

Aviso: Al utilizar la clase Scanner para leer los datos, éstos se leen localizados. Esto significa que para introducir un número real, por ejemplo el coste por litro, hay que escribirlo con coma decimal, no con punto decimal. Si se hace con punto decimal dará un error al intentar convertirlo a un número válido. Además, si en un momento que se esperaba un número entero, se introduce otro tipo de dato, el sistema termina abruptamente.

Ejercicio 1.24:

Escriba un programa que calcule cuánto le dará su banco después de realizar una imposición a plazo fijo. Para ello el programa debe pedir la cantidad que desea invertir en el banco, el tipo de interés anual que le paga el banco por el dinero y el plazo que se mantiene la inversión. El programa debe calcular el dinero que se obtiene después de dicho plazo. Recuerde que al pagarle los intereses el banco le retendrá el 18% para hacienda. Escriba los mensajes apropiados para que el usuario pueda seguir el proceso de cálculo realizado.

Planteamiento: Hay que realizar un programa que utilice variables donde se vayan recogiendo los valores que se piden al usuario, que serán el capital a invertir (capitalInvertido), el tipo de interés anual (interésAnual) y los meses que se mantiene la inversión (mesesInversión). El dato de la retención que se aplica sobre los intereses es fija del 18%, por lo que se definirá como una constante (RETENCION). Se declarará un objeto de la clase Scanner para solicitar al usuario los valores y leerlos. Después se realizarán los cálculos y se imprimirán por pantalla los resultados.

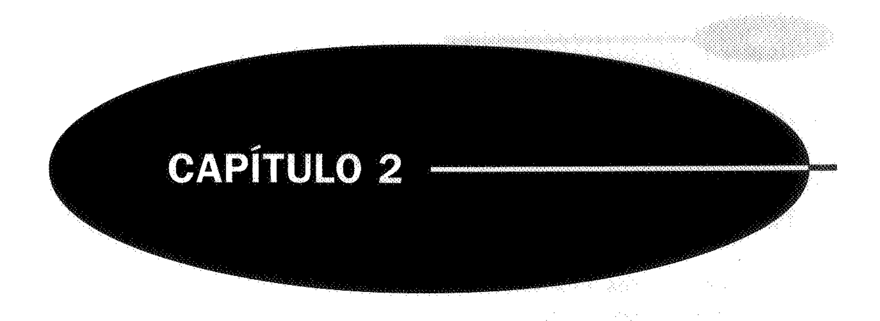
Solución:

```
import java.util.*;
public class CalculoInteresesUsuario {
    public static void main(String[] args) {
      final double RETENCION = 18;
      final int MESES_AÑO = 12;
      Scanner teclado = new Scanner(System.in);
      System.out.print("Introduzca la cantidad que desea invertir: ");
      double capitalInvertido = teclado.nextDouble();
      System.out.print("Introduzca el interés anual que le ofrece el banco: ");
      double interésAnual = teclado.nextDouble();
      System.out.print("Introduzca el número de meses de la inversión: ");
      int mesesInversión = teclado.nextInt();

      double interesesObtenidos = capitalInvertido *
                                  interésAnual / 100 *
                                  mesesInversión / MESES_AÑO;
      double retención = interesesObtenidos * RETENCION / 100;
      double interesesCobrados = interesesObtenidos - retención;

      System.out.println();
      System.out.println("Cálculo de intereses.");
      System.out.printf("Dinero a invertir: %.2f€.\n", capitalInvertido);
      System.out.printf("Interés anual: %.2f%%.\n", interésAnual);
      System.out.printf("Intereses a los %d meses: %.2f€.\n",
                            mesesInversión, interesesObtenidos);
      System.out.printf("Retención que se realiza: %.2f€.\n", retención);
      System.out.printf("Intereses a cobrar: %.2f€.\n", interesesCobrados);
    }
}
```

Aviso: Recuerde que al utilizar la clase Scanner los datos se leen localizados, por lo que se debe introducir los valores no enteros con coma decimal.



Clases y objetos

2.1 ESTRUCTURA DE UNA CLASE

Como ya se ha comentado en el Capítulo 1, la estructura de una clase suele seguir el siguiente esquema:

```
/**
 * Estructura de una clase en Java
*/
public class NombreDeClase {
    // Declaración de los atributos de la clase

    // Declaración de los métodos de la clase

    // El método main, que indica donde empieza la ejecución
    public static void main(String[] args) {
        // Declaración de las variables del método

        // Sentencias de ejecución del método

    }
}
```

En el caso de ser una clase distinta de la clase principal, el método `main()` no suele aparecer. Una clase es la descripción (modelo) de un tipo de objetos. Una aplicación, un programa, se compone de un conjunto de clases, que crean objetos que interactúan entre sí. El nombre de la clase se empezará, como convenio de programación, en mayúscula. Una vez que la clase está disponible, el programador puede instanciar, crear, objetos de dicha clase. Por ello, los términos "objeto" e "instancia de clase" o, simplemente, "instancia" se usan como sinónimos.

Para disponer de un objeto de la clase hace falta declarar una variable y crear el objeto. Suponga que dispone de una clase `Alumno` de la siguiente forma, donde se ha suprimido su contenido:

```
class Alumno {  }
```

La declaración de variables es similar a la de una variable, como se ha hecho en el Capítulo 1.

```
Alumno alumno1;
Alumno alumno2;
```

Para crear un objeto para cada variable, se utiliza el operador new:

```
alumno1 = new Alumno();
alumno2 = new Alumno();
```

Referencia null
Una referencia a un objeto puede no tener asignada ninguna instancia. Existe un valor especial, llamado null, que indica cuándo una referencia no tiene asignada ninguna instancia. Como ejemplo, se declara un nuevo alumno3 de la clase Alumno:

```
Alumno alumno3; // vale "null" por defecto
```

Se puede poner explícitamente una referencia a null:

```
alumno2 = null;  // vale "null" por asignación explícita
```

Cuando una referencia a un objeto vale null, no se puede utilizar como objeto, pues no existe como tal.

Referencias compartidas: alias
Es posible que se disponga de varias referencias a un mismo objeto. Si sobre las declaraciones anteriores de alumno1 y alumno2 se hace:

```
alumno1 = alumno2;      // asignación de referencias
```

en este momento, las variables alumno1 y alumno2 hacen referencia al mismo objeto de la clase alumno. Ello implica que cualquier modificación del objeto alumno1 modifica también el objeto al que hace referencia alumno2, ya que realmente es el mismo.

2.2 ATRIBUTOS

Los atributos permiten guardar la información de un objeto. Por ejemplo, para un alumno se necesita saber la siguiente información: el nombre, los apellidos, el curso en que está matriculado, si su horario es de mañana o de tarde, etc. Estos datos se almacenan en campos o atributos que se declaran como variables en el ámbito de la clase. La declaración de atributos se hace de la siguiente forma:

```
enum Horario { MAÑANA, TARDE } // posibles horarios

class Alumno {
    String nombre;
    String apellidos;
    int añoDeNacimiento;
    int númeroPersonal;   // Número Personal: identificativo único
    String grupo;
    Horario horario = Horario.MAÑANA;
}
```

En este ejemplo, se ha declarado que la clase Alumno contiene seis atributos, nombre, apellidos, añoDeNacimiento, númeroPersonal, grupo y horario. El Horario se ha definido como un enumerado con dos posibles turnos, MAÑANA y

TARDE. Como puede observar se puede dar un valor inicial a los atributos en la declaración. De esta forma se ha indicado que el horario predeterminado para un alumno es el de mañana.

Si los atributos no se declaran como privados ni protegidos (consulte el Capítulo 3), se puede acceder a sus valores con la notación objeto.atributo. Por ejemplo, para imprimir por pantalla el nombre del alumno1, se puede escribir:

```
System.out.println("El nombre es: " + alumno1.nombre);
```

Aunque es posible acceder a un atributo de la forma indicada, suele ser preferible acceder a los mismos mediante un método, llamado en este caso método de acceso.

⇨ Sobre los atributos consulte los Ejercicios 2.1 y 2.2.

2.3 MÉTODOS

Los métodos sirven para definir el comportamiento del objeto en sus interacciones con otros objetos. Siguiendo el ejemplo de objetos de la clase Alumno se puede pedir su nombre, asignarle grupo, etc.

```
enum Horario { MAÑANA, TARDE } // posibles horarios

class Alumno {
    String nombre;
    String apellidos;
    int añoDeNacimiento;
    int númeroPersonal;  // Número Personal: identificativo único
    String grupo;
    Horario horario = Horario.MAÑANA;

    public String dameGrupo() { ... }
    public void ponGrupo(String nuevoGrupo) { ... }
    ...
}
```

Se han añadido dos métodos a la clase Alumno, uno llamado dameGrupo() para pedir a un objeto de la clase Alumno el grupo al que asiste a clase y otro ponGrupo() para poder asignar un nuevo grupo a un Alumno. Se ha omitido por claridad el cuerpo con el código de los métodos.

Para utilizar un método de la clase se utiliza la notación objeto.metodo(), pasando entre los paréntesis los argumentos que necesite el método al que se llama. Por ejemplo, para dar al alumno1 el grupo "INF-01" y luego imprimir por pantalla el grupo que tiene asignado, solicitándoselo mediante su método dameGrupo(), se puede escribir:

```
alumno1.ponGrupo("INF-01");
System.out.println("El alumno está en el grupo: " + alumno1.dameGrupo());
```

Además de utilizar los nombres ponXXX() para los métodos que asignan un valor a un atributo y dameXXXX() para los métodos que devuelven el valor del atributo, suele ser común utilizar la forma inglesa setXXXX() y getXXXX(), respectivamente. En los ejemplos de los distintos capítulos, se usará una u otra indistintamente, prefiriéndose la versión inglesa en los capítulos avanzados.

Valor de retorno

Los métodos pueden realizar una función y no devolver un valor, lo que se indica con la cláusula void, o pueden comportarse como una función y devolver un valor primitivo o una referencia a un objeto. En este caso, se pone

delante del nombre del método el tipo o clase del valor devuelto. En el caso del método dameGrupo(), el método devuelve una referencia a un objeto de la clase String. Para devolver un valor dentro del método, se utiliza la sentencia return. El método dameGrupo() se puede escribir:

```
public String dameGrupo(){
   return grupo;
}
```

Autoreferencia this
Para referirse a los atributos del objeto desde un método del mismo, se puede hacer directamente con su nombre o utilizando this. Esta palabra del lenguaje se utiliza, sobre todo, cuando existe ambigüedad entre nombres de parámetros de un método y atributos del objeto (otros usos se explican más adelante). Por ejemplo, en el siguiente método

```
public void ponGrupo(String grupo, Horario horario) {
   this.grupo   = grupo;
   this.horario = horario;
}
```

this funciona como una referencia especial, de forma que this.grupo se refiere al atributo grupo declarado en la clase, para diferenciarlo de la variable grupo declarado como parámetro del método.

Sobrecarga
La única limitación en la elección del nombre de un método es que, en una clase, todos los métodos deben tener diferente signatura (básicamente distinto nombre y parámetros). Esto permite que existan varios métodos con el mismo nombre pero con diferentes parámetros. Por ejemplo, se podrían tener dos métodos de nombre ponGrupo().

```
public void ponGrupo(String grupo, Horario horario){
   this.grupo   = grupo;
   this.horario = horario;
}

public void ponGrupo(String grupo){
   this.grupo   = grupo;
}
```

Dependiendo de los valores de los argumentos con que se llame al método ponGrupo, se ejecutaría uno u otro de los definidos.



Sobre la definición y uso de métodos consulte los Ejercicios a partir del 2.2.
Sobre la sobrecarga de métodos consulte los Ejercicios 2.17, 2.18.

2.4 CONSTRUCTORES

Al principio del capítulo se ha visto que para crear un objeto se usa la instrucción new seguida del nombre de la clase y una pareja abre paréntesis - cierra paréntesis:

```
Alumno alumno1= new Alumno();
```

Esta operación invoca al constructor por defecto, que se proporciona automáticamente y tiene el mismo nombre que la clase.

Lo habitual es que al escribir una clase se desee construir los objetos de la clase de una determinada forma. Para ello se escribe uno o más constructores. Para definir un constructor, se pone el tipo de acceso, el nombre de la clase, los parámetros que acepta, si lanza excepciones (opcional) y un cuerpo o bloque de código, de la forma:

```
acceso nombreClase (parámetros) throws excepciones { cuerpo }
```

Así, para la clase Alumno, se pueden proporcionar dos constructores: uno que recibe el nombre, los apellidos y el año de nacimiento; y otro que, además, aceptase el grupo y el horario. El código sería el siguiente:

```
class Alumno {
    Alumno(String nombre, String apellidos, int año) {
        this.nombre    = nombre;
        this.apellidos = apellidos;
        this.añoDeNacimiento = año;
    }

    Alumno(String nombre, String apellidos, int añoDeNacimiento,
           String grupo, Horario horario) {
        this.nombre    = nombre;
        this.apellidos = apellidos;
        this.añoDeNacimiento= añoDeNacimiento;
        this.grupo    = grupo;
        this.horario = horario;
    }

    // resto de la clase, igual
}
```

Los constructores declarados sustituyen al proporcionado por defecto. Por tanto, el constructor por defecto ya no se puede utilizar. Ahora, para construir cualquier alumno es necesario proporcionar los valores apropiados de acuerdo con el constructor que se utilice:

```
Alumno alumno1= new Alumno("Juan", "García", 1980);
Alumno alumno2= new Alumno("María", "López", 1981, "INF-01", Horario.TARDE);
```

Ahora sería ilegal el siguiente código:

```
Alumno alumno2= new Alumno(); // error: no hay constructor sin parámetros
```

El uso de los constructores permite comprobar que los objetos se construyen apropiadamente con valores correctos. Si los valores pasados al constructor no son apropiados, se puede lanzar una excepción para no crear el objeto. Para conocer más sobre el uso y manejo de excepciones, consulte el Capítulo 4.

```
class Alumno {
    Alumno(String nombre, String apellidos, int año) throws Exception {
        // Si el nombre o los apellidos son cadenas null
        // se lanza una excepción y el objeto no se crea
        if (nombre == null || apellidos == null)
            throw new Exception("Argumentos no válidos");

        // Si el año es negativo
```

```
            // se lanza una excepción y el objeto no se crea
            if (año < 0)
                throw new Exception("Año incorrecto");

            this.nombre= nombre;
            this.apellidos= apellidos;
            this.añoDeNacimiento= año;
        }
        // resto de la clase, igual
    }
```

⇨ Sobre la definición de constructores consulte los Ejercicios 2.8, 2.10 y 2.18.

Autoreferencia `this`

En la construcción de objetos se puede utilizar un constructor ya definido desde otro constructor. Para ello se utiliza la referencia `this()` seguida de los parámetros del constructor que se desea invocar. Esta llamada sólo se puede realizar como primera sentencia de un método. De esta forma, los constructores de la clase se pueden escribir:

```
class Alumno {
    Alumno(String nombre, String apellidos, int año) throws Exception {
        // Si el nombre o los apellidos son cadenas null
        // se lanza una excepción y el objeto no se crea
        if (nombre == null || apellidos == null)
            throw new Exception("Argumentos no válidos");

        // Si el año es negativo
        // se lanza una excepción y el objeto no se crea
        if (año < 0)
            throw new Exception("Año incorrecto");

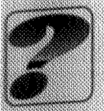
        this.nombre= nombre;
        this.apellidos= apellidos;
        this.añoDeNacimiento= año;
    }

    // constructor de la clase alumno con grupo y horario
    // Se invoca al constructor anterior para comprobar y poner
    // el nombre, apellidos y año de nacimiento
    Alumno(String nombre, String apellidos, int añoDeNacimiento,
            String grupo, Horario horario) throws Exception {
        this(nombre, apellidos, añoDeNacimiento);

        if (grupo == null)
            throw new Exception("Grupo no especificado");

        this.grupo= grupo;
        this.horario= horario;
    }

    // resto de la clase, igual
}
```



Problemas resueltos

Atributos

Ejercicio 2.1:

Definir una clase que represente a un coche. En la definición se debe incluir:

- *el modelo,*
- *el color,*
- *si la pintura es metalizada o no,*
- *la matrícula,*
- *el tipo de coche, que puede ser MINI, UTILITARIO, FAMILIAR o DEPORTIVO*
- *el año de fabricación*
- *la modalidad del seguro, que puede ser a terceros o a todo riesgo*

Planteamiento: Al diseñar una clase, hay que prestar especial atención a los atributos que se incluyen y sus tipos. Para los atributos hay que elegir nombres significativos, por ejemplo, "`modelo`" es un buen nombre para el atributo que representa el modelo del coche. Al elegir el tipo de un atributo, se debe pensar tanto en los valores que puede tomar, como en las operaciones que se van a aplicar.

Para el modelo, el color y la matrícula, se elige el tipo `String`. Para determinar condiciones que se satisfacen o no, se debe elegir un `boolean`, como es el caso de si la pintura es metalizada o no.

Cuando un atributo puede tomar una serie de valores, lo más adecuado es utilizar un tipo enumerado, eligiendo los nombres adecuados para los diferentes valores. De esta forma, para el tipo de coche se usa un enumerado con los valores que sugiere el enunciado.

Para el año de fabricación, se usa un número entero. Evidentemente, valores negativos o muy grandes podrían no ser legales, pero queda a cargo del programador comprobarlo.

Por último, para la modalidad de seguro, de nuevo es adecuado un enumerado, con los valores `{ A_TERCEROS, A_TODO_RIESGO }`.

Solución:

```
enum TipoDeCoche { MINI, UTILITARIO, FAMILIAR, DEPORTIVO };

enum TipoDeSeguro { A_TERCEROS, A_TODO_RIESGO };

public class Coche {

    String modelo;
    String color;
    boolean esMetalizado;
    String matricula;
    TipoDeCoche tipo;
    int añoDeFabricación;
    TipoDeSeguro seguro;

}
```

Los nombres de los atributos han de ser lo más significativos posible, pero no han de ser demasiado largos.

Escoger adecuadamente los tipos de los atributos no siempre es tarea fácil.

Comentario: Se podría pensar en utilizar un valor de tipo `boolean` para la modalidad de seguro, por ejemplo, `boolean seguroATerceros`. Se usaría el valor `true` cuando se disponga de un seguro a terceros, y `false` cuando el seguro sea a todo riesgo. En general, no es una buena decisión usar los booleanos para atributos que toman

dos valores. Por un lado, no queda claro el significado del valor `false`. Por otro, se pueden aplicar operaciones que quizá no tengan sentido. Lo peor es que compromete la extensibilidad del programa: si se necesitara otra modalidad de seguro habrá que modificar muchas líneas de código, mientras que añadir un valor nuevo a un enumerado es sumamente sencillo.

Aviso: En este problema se ha abordado la elección de los atributos y sus tipos, sin tener en cuenta el tipo de acceso de los mismos; esto se tratará en posteriores ejercicios.

Ejercicio 2.2:

Se desea imprimir el modelo y el color de un coche dado. Para ello, se pide escribir un método, que acepte un objeto de la clase Coche. Si dicha referencia no es null, el método deberá imprimir el modelo y el color. Si es null, el método no hará nada.

Planteamiento: No se dice el nombre del método, por lo que se elige uno significativo del problema, por ejemplo, `imprimeCoche`. Este método acepta una referencia de la clase `Coche`. Se usa una sentencia `if` para comprobar si la referencia es `null` o contiene un objeto, en cuyo caso se imprime el modelo y el color.

Parámetros: La referencia a un objeto de la clase `Coche`.

Valor de retorno: El método no necesita devolver ningún valor de retorno por lo que se utiliza la cláusula `void`.

Solución:

```
public void imprimeCoche (Coche coche) {

    if (coche != null) {
        System.out.println("El coche " + coche.modelo +
                           " es de color " + coche.color);
    }
}
```

Para acceder al valor de los atributos, se pone el nombre del objeto, un punto y el nombre del atributo.

Comentario: En el ejercicio siguiente se verá una forma más conveniente de escribir métodos que añaden comportamiento a una clase.

Aviso: Es un error común que en ejecución se use un método o atributo de una referencia que está a `null`. Esto provoca un error en tiempo de ejecución (`RunTimeError`) denominado `NullPointerException`, que normalmente terminará la ejecución del programa. Es una buena práctica asegurarse de que al usar una referencia, no contendrá el valor especial `null`.

MÉTODOS

Ejercicio 2.3:

Añadir a la clase Coche del Ejercicio 2.1 un método de nombre imprimeCoche que imprima el modelo y el color del coche.

Planteamiento: Cuando se invoca un método de un objeto, desde el mismo se puede acceder directamente a los atributos de la clase. Por tanto, simplemente se han de imprimir los atributos `modelo` y `color`.

Parámetros: Ninguno. Al ser un método de la clase `Coche`, no necesita ningún valor como parámetro ya que se dispone del modelo y color del coche como atributos.

Valor de retorno: El método no necesita devolver ningún valor de retorno por lo que se utiliza la cláusula `void`.

Solución:

```
enum TipoDeCoche { MINI, UTILITARIO, FAMILIAR, DEPORTIVO };

enum TipoDeSeguro { A_TERCEROS, A_TODO_RIESGO };

class Coche {

    String modelo;
    String color;
    boolean esMetalizado;
    String matricula;
    TipoDeCoche tipo;
    int añoDeFabricación;
    TipoDeSeguro seguro;

    public void imprimeCoche () {
      System.out.println("El coche " + modelo +
                         " es de color " + color);
    }
}
```

El comportamiento de la clase Coche se recoge en su definición.

Los atributos del objeto dentro de la misma clase se usan directamente, sin cualificación.

Comentario: Este ejercicio es similar al anterior en cuanto a funcionalidad. Sin embargo, el comportamiento de los objetos ha de recogerse siempre en la definición de la clase y no depender de funciones externas en la medida de lo posible.

Ejercicio 2.4:

Escribir un programa que tenga una instancia de mi coche, que es un Rolls Royce de color dorado. El programa ha de imprimir un mensaje que diga de qué modelo y color es mi coche.

Planteamiento: Los objetos han de usarse por medio de referencias. Por tanto, habrá que declarar una referencia a la clase Coche. Además, hay que elegir un nombre significativo para la referencia. Como en este caso, se guarda la información de mi coche, el nombre miCoche (u otro similar) puede ser adecuado. Luego se inicializan los atributos modelo y color. Para ello, se escribe el nombre de la referencia, un punto, y el nombre del atributo, de la forma miCoche.modelo. Y con el operador de asignación ("=") se le asigna el valor "Rolls Royce". Lo mismo aplica al atributo color. Por último, se imprimen esos valores; como la clase Coche ya dispone del método imprimeCoche() (del ejercicio anterior), se puede invocar dicho método de la forma miCoche.imprimeCoche().

Solución:

```
enum TipoDeCoche { MINI, UTILITARIO, FAMILIAR, DEPORTIVO };

enum TipoDeSeguro { A_TERCEROS, A_TODO_RIESGO };

class Coche {

    String modelo;
    String color;
    boolean esMetalizado;
    String matricula;
    TipoDeCoche tipo;
    int añoDeFabricación;
```

```
    TipoDeSeguro seguro;

    public void imprimeCoche () {
        System.out.println("El coche " + modelo +
                           " es de color " + color);
    }
}

class pruebaCoche {
    public static void main (String args[]) {
        Coche miCoche= new Coche();

        miCoche.modelo= "Rolls Royce";
        miCoche.color= "dorado";

        miCoche.imprimeCoche();
    }
}
```

Se invoca a cierto comportamiento de los objetos de tipo Coche.

Comentario: La salida por pantalla que produce el programa es:

```
El coche Rolls Royce es de color dorado
```

Ejercicio 2.5:

Escriba una clase que represente un Semáforo, que podrá estar rojo, ámbar o verde. La clase tendrá un atributo llamado color, inicialmente a rojo. También dispondrá de un atributo que determine si el semáforo está parpadeando. Inicialmente, el semáforo no está parpadeando.

Planteamiento: Para describir el color del semáforo, se usa un tipo enumerado con los tres posibles colores denominado ColorSemáforo. Después, se escribe la clase Semáforo, que tendrá dos atributos: el primero será el color, de tipo ColorSemáforo; el segundo representará si el semáforo está parpadeando, para lo que se usa un boolean.

Solución:

```
enum ColorSemáforo { ROJO, ÁMBAR, VERDE };

public class Semáforo {

   ColorSemáforo color = ColorSemáforo.ROJO;
   boolean estáParpadeando = false;

}
```

Este enumerado representa el color que puede tomar un semáforo.

Se pueden inicializar los atributos de un objeto en la misma línea que se declaran.

Comentario: Los atributos de los objetos pueden tomar un valor inicial en la misma línea que se declaran. No obstante, lo normal es que cada objeto tenga valores distintos, en cuyo caso se proporcionará uno o más constructores para el valor inicial de los atributos de un objeto, como se verá en siguientes ejercicios.

Aviso: Para los atributos que especifican condiciones que se cumplen o no, es muy habitual utilizar el tipo boolean. Los nombres de dichos atributos suelen ser adjetivos o, como en el ejemplo, una palabra formada por el verbo "estar" y la condición.

Ejercicio 2.6:

Escriba un programa que instancie un semáforo de la clase Semáforo del ejercicio anterior. El programa escribirá por pantalla el color del semáforo. Luego pondrá el semáforo en ámbar y volverá a imprimir el color.

Planteamiento: Se define una clase para el método `main()`, que se llamará `PruebaSemáforo`. En dicho método `main()`, se declara un semáforo, llamado `s1`, por ejemplo. Se instancia mediante `new` un objeto de la clase `Semáforo` y se asigna a `s1`. Ahora se puede imprimir su color, luego se cambia a ámbar y se vuelve a imprimir.

Solución:

```
enum ColorSemáforo { ROJO, ÁMBAR, VERDE };

class Semáforo {

    ColorSemáforo color = ColorSemáforo.ROJO;
    boolean estáParpadeando = false;
}

class PruebaSemáforo {

    public static void main (String args[]) {
        Semáforo s1;

        s1 = new Semáforo(); ←

        System.out.println("S1 está " + s1.color);

        s1.color = ColorSemáforo.ÁMBAR; ←
        System.out.println("S1 está " + s1.color);
    }
}
```

Se crea el objeto semáforo con la instrucción new.

Para poner un valor de un tipo enumerado, se escribe el nombre del tipo, un punto y el valor deseado.

Comentario: Hasta que no se genera el objeto con `new`, la referencia `s1` contiene el valor `null` (inicialización por defecto de Java). Es muy común hacer ambas cosas en la misma línea, como se presenta a continuación:

```
Semáforo s1 = new Semáforo();
```

Aviso: La forma utilizada en la solución para acceder a los atributos no se considera la más adecuada. Ya se verá en otros ejercicios los métodos accesores y los derechos de acceso. En el siguiente capítulo se hará hincapié en el tema.

Ejercicio 2.7:

Escriba un programa que disponga de una clase para representar las asignaturas de una carrera. Una asignatura tiene un nombre, un código numérico y el curso en el cual se imparte. Los valores iniciales han de proporcionarse en el constructor. La clase ha de tener métodos para obtener los valores de los atributos. El programa ha de construir un objeto con los siguientes valores: nombre "Matemáticas", código 1017, curso 1. A continuación, el programa ha de imprimir los valores del objeto por pantalla.

Planteamiento: Se proporcionarán dos clases, una que representa a las asignaturas, llamada `Asignatura`, y otra para el programa, llamada `EjercicioAsignatura`. La clase `Asignatura` dispondrá de un constructor que acepta los valores para representar una asignatura: el nombre, el código y el curso. El nombre del constructor

es el mismo de la clase, Asignatura en este caso. Los atributos serán privados y se disponen de métodos para acceder a sus valores. Normalmente, estos métodos denominados accesores, se suelen llamar como el atributo, anteponiendo el verbo dame o get. En la clase EjercicioAsignatura, habrá un método main(), en el que se crea la asignatura de matemáticas de primero y luego una sentencia para imprimir sus datos.

Solución:

```
class Asignatura {

    private String nombre;
    private int código;   <-- Conviene que los atributos sean privados y accesibles con métodos explícitos.
    private int curso;

    public Asignatura (String nombre, int código, int curso) {
        this.nombre = nombre;
        this.código = código;
        this.curso  = curso;
    }

    public String dameNombre(){   <-- Métodos accesores a los atributos. Típicamente estos métodos se llaman dameAtributo o getAtributo.
        return nombre;
    }

    public int dameCódigo(){
        return código;
    }

    public int dameCurso(){
        return curso;
    }
}

class EjercicioAsignatura {

    public static void main(String args[]) {
        Asignatura a = new Asignatura ("Matemáticas", 1017, 1);

        System.out.println("Asignatura " + a.dameNombre() +
                           " código "    + a.dameCódigo() +
                           " del curso " + a.dameCurso());
    }
}
```

Comentario: Se podría haber incluido el método main() en la clase Asignatura, con lo que con una sola clase se hubiera resuelto el ejercicio. Es muy típico añadir un main() a clases intermedias para probar o ejercitar la clase, aunque dicho método main() no forme parte del proyecto en el que participa la clase.

Respecto de los métodos accesores, nótese que no disponen de parámetros, lo que se indica con los paréntesis vacíos, como en dameNombre(). Estos métodos devuelven el mismo tipo que el atributo al que acceden.

Aviso: La codificación de los métodos accesores es tremendamente trivial, lo que suscita la tentación de eliminarlos y poner los atributos como públicos. Se desaconseja esta práctica, pues el programador ha de responsabilizarse de una clase y no debe permitir que se modifiquen los atributos de forma arbitraria.

Ejercicio 2.8:

Definir una clase que represente un punto en un espacio bidimensional. La clase debe disponer de un constructor con las coordenadas del punto y métodos accesores a las coordenadas.

Planteamiento: En un espacio bidimensional, un punto se representa por un par ordenado de números reales. Se deben elegir nombres representativos para los mismos, como `x` e `y`. El tipo más adecuado dependerá del tipo de aplicación en que se usen estos objetos. Como el enunciado no dice nada, se usará `double`. Para asegurar la consistencia de un objeto, es conveniente que los atributos sean privados y existan métodos públicos para averiguar su valor. Estos métodos se llamarán `dameX()` y `dameY()`, sin parámetros, y con tipo de retorno igual al del atributo que devuelven.

Solución:

```
public class Punto {

    private double x, y;

    public Punto(double x, double y) {
        this.x = x;
        this.y = y;
    }

    public double dameX(){
        return x;
    }

    public double dameY(){
        return y;
    }
}
```

Los atributos son privados, de forma que sólo pueden modificarse mediante métodos de la clase.

Nótese el uso de `this` para resolver la ambigüedad entre los nombres de los parámetros y los atributos.

Los métodos accesores proporcionan la información adecuada sobre el objeto.

Comentario: Fíjese que se ha elegido `double` como tipo para las coordenadas. No obstante, ciertas aplicaciones (por ejemplo, gráficos) podrían requerir `int`, o podría ser suficiente `float`.

Aviso: Se ha elegido representar el punto en coordenadas cartesianas. Se podría representar en polares, lo que supondría tener que programar las conversiones entre los diferentes sistemas de representación.

Ejercicio 2.9:

Escriba un programa que instancie cuatro puntos: el primero situado en el origen, el segundo situado en (5, 3), el tercero en (2, -1) y para el cuarto como punto medio entre el segundo y el tercero.

Planteamiento: Los tres primeros puntos se crean con los valores literales que nos proporciona el enunciado. Para el último, se calculan sus coordenadas como las medias entre las respectivas coordenadas del segundo y tercer puntos.

Solución:

```
public static void main (String args[]) {
    Punto p1 = new Punto (0, 0); // origen de coordenadas

    Punto p2 = new Punto (5, 3);
    Punto p3 = new Punto (2, -1);
```

Ejemplos de creación de objetos mediante new.

```
        Punto p4 = new Punto ((p2.dameX() + p3.dameX())/2,
                              (p2.dameY() + p3.dameY())/2);
    }
```

Comentario: Los tres primeros puntos se crean con los valores literales que nos indica el enunciado. Para el cuarto punto, se usan los métodos accesores de los puntos, obteniendo las coordenadas de los puntos segundo y tercero, haciendo luego la semisuma.

Para crear objetos, se usa la instrucción `new` seguida del nombre de la clase y después, entre paréntesis, los parámetros para el constructor de la clase.

Ejercicio 2.10:

Añada a la clase Punto un constructor sin parámetros que permita construir puntos en el origen de coordenadas.

Planteamiento: Se pide que los valores de las coordenadas sean (0, 0). Se pueden inicializar directamente, pero es más conveniente invocar al otro constructor con los argumentos a 0. De esta forma, se dispone de un solo constructor que realiza el código de inicialización. Para invocar a otro constructor, se utiliza el nombre especial `this`, seguido de los parámetros que espera ese constructor. En este caso, se escribe `this(0,0)`. Esta invocación ha de aparecer como la primera sentencia del cuerpo del constructor.

Solución:

```
// sólo se incluye el constructor, el resto del código como
// el ejercicio anterior
public Punto () {
    this (0, 0);
}
```

Se invoca al constructor anterior con los valores deseados.

Comentario: Es muy adecuado tener diferentes constructores cuando se desea que el objeto tenga valores iniciales tomados de diferente forma.

Aviso: Cuando los diferentes constructores tienen código común, es conveniente que unos se llamen a otros, si es posible, en lugar de repetir código.

Ejercicio 2.11:

Añada un método a la clase Punto que calcule la distancia a otro punto.

Planteamiento: El método debe recibir como parámetro el punto al cual se calcula la distancia. Se aplica la fórmula de la distancia euclídea entre dos puntos con ayuda de la clase `Math`. En dos variables auxiliares se almacenan las diferencias entre las coordenadas de los puntos.

Parámetros: El punto al cual se calcula la distancia, es decir, un objeto de la clase `Punto`.

Devuelve: Un `double` con el valor de la distancia al punto pasado como parámetro.

Solución:

```
// sólo el método distancia
public double distancia (Punto p) {
    double diffX = x - p.x;
    double diffY = y - p.y;

    return Math.sqrt (diffX * diffX + diffY * diffY);
}
```

$$d = \sqrt{(a_x - b_x)^2 + (a_y - b_y)^2}$$

Comentario: Para elevar un número al cuadrado, se podría haber optado por la función pow, escribiendo alternativamente:

```
return Math.sqrt (Math.pow (diffX, 2) + Math.pow (diffY, 2));
```

Aviso: El método distancia puede fallar si se le pasa un punto a null. En efecto, si p es null, no se puede acceder a los atributos.

Ejercicio 2.12:

Modificar el método distancia del ejercicio anterior para que lance una excepción si el punto que se le pasa es null. La excepción deberá contener un mensaje descriptivo del problema.

Planteamiento: Se debe modificar la cabecera del método distancia para declarar que ahora va a lanzar excepciones. Para ello, después de los parámetros se añade la palabra reservada throws seguida del tipo de excepción que se va a lanzar. En este caso, se usa Exception. Con un if se comprueba si la referencia p contiene el valor null, en cuyo caso se lanza una excepción. En caso contrario, p contiene un objeto y puede accederse a sus atributos.

Parámetros: El punto al cual se calcula la distancia, es decir, un objeto de la clase Punto.

Devuelve: Un double con la distancia al punto pasado como parámetro.

Solución:

```
// sólo el método distancia
public double distancia (Punto p) throws Exception  {
    if(p == null)
        throw new Exception("p no puede valer null");

    double diffX = x - p.x;
    double diffY = y - p.y;

    return Math.sqrt(diffX * diffX + diffY * diffY);
}
```

El método distancia comprueba que su(s) argumento(s) son correctos.

Comentario: Es muy recomendable que los métodos comprueben primero los valores que se les pasan y, si no están en un rango aceptable, se lance una excepción, indicando la naturaleza del problema. A la larga, esto facilita tanto el descubrimiento y diagnosis de errores en programas grandes, como el mantenimiento del programa.

Ejercicio 2.13:

Escriba un programa que cree un punto en (4, 3) e imprima la distancia del punto al origen de coordenadas.

Planteamiento: Para calcular la distancia de un punto al origen, se aplica el teorema de Pitágoras, resultando $d = \sqrt{x^2 - y^2}$. No obstante, dado que se dispone del método distancia a otro punto, se puede calcular la distancia al punto origen de coordenadas.

Escribe: Se escribe el mensaje "Distancia de p al origen" seguido del valor.

Solución:

```
public class distanciaAlOrigen {

    public static void main (String args[]) {
```

```
        // origen de coordenadas
        Punto origen = new Punto (0, 0);

        Punto p = new Punto (4, 3);

        System.out.println("Distancia de p al origen " + p.distancia(origen));
    }
}
```

Se utiliza el método distancia a otro punto para hacer el cálculo.

Aviso: Es conveniente estudiar el API de Java y las bibliotecas disponibles para evitar escribir varias veces el mismo código.

Ejercicio 2.14:

Escriba un método llamado `sumaTotal` que, dada una lista arbitraria de números enteros, devuelva la suma de todos ellos. Además, escriba un programa que pruebe dicho método, calculando la suma de las secuencias {3, 5, 2, 4, 6}, la secuencia {2, 10, -1, 2}, la secuencia {10} y la secuencia {}. El programa imprimirá el valor de dichas sumas.

Planteamiento: El método `sumaTotal` deberá aceptar un número variable de parámetros, lo que se indica con la notación "...". Entonces, deberá ir recorriendo todos los parámetros e ir acumulando la suma de cada uno de ellos. Para ello, se guarda en una variable llamada `suma` el resultado parcial.

El valor inicial de la variable `suma` será 0. Para sumar los valores que se reciben como parámetro, se usa un bucle `for` sobre la lista de valores recibidos en el método. Por último, el método devolverá el valor almacenado en `suma`.

Parámetros: La lista de números enteros, denotado como `int ... números`.

Valor de retorno: El valor de la suma acumulada. Como la suma de enteros es un entero, el valor de retorno será del tipo `int`.

Solución:

```
public class suma {

    public static int sumaTotal (int ... números) {
        int suma = 0;

        for (int num: números) {
            suma += num;
        }

        return suma;
    }

    public static void main (String args[]) {
        int x;

        x = sumaTotal (3, 5, 2, 4, 6);
        System.out.println("Primera suma = " + x);

        x = sumaTotal (2, 10, -1, 2);
        System.out.println("Segunda suma = " + x);
```

La notación ... significa que se espera un número arbitrario (0 o más) de parámetros.

Con un bucle `for` se recorren los parámetros con los que se llama al método.

```
        x = sumaTotal (10);
        System.out.println("Tercera suma = " + x);

        x = sumaTotal ();
        System.out.println("Cuarta suma = " + x);
    }
}
```

Comentario: Este programa imprimirá por pantalla lo siguiente:

```
Primera suma = 20
Segunda suma = 13
Tercera suma = 10
Cuarta suma = 0
```

Hay que resaltar la cuarta suma: cuando se llama al método `sumaTotal`, se pasa una lista (array, véase **Capítulo 6**) vacía, con 0 elementos. El cuerpo del bucle `for`, por tanto, no se llega a ejecutar ninguna vez, con lo que la variable `suma` no se modifica.

Aviso: Hay que tener especial cuidado con los métodos con un número variable de argumentos cuando la lista está vacía. Por otro lado, si se desea disponer de un número variable de argumentos, la variable para ellos debe ser la última de los parámetros que acepta el método.

Ejercicio 2.15:

Escriba un método de nombre mediaAritmetica que, dada una lista de números reales, calcule y devuelva su media aritmética. Escriba también un programa que, utilizando dicho método, determine la media de los números -10, 5, 0, 7, 20.

Planteamiento: La media aritmética de una serie de números se calcula como $\frac{\sum_{i=1}^{n} x_i}{n}$.

Por tanto, hay que calcular la suma de todos los números, de forma similar al ejercicio anterior: para ello se recorre la lista de los números y se determina su suma. Posteriormente, se divide entre el número de valores en la lista, información que se puede obtener del atributo `length` de la secuencia.

Parámetros: La lista de valores, denotado como `double... números`.

Valor de retorno: La media aritmética de valores `double` será un valor del tipo `double`.

Solución:

```
public class mediaAritmetica {

    public static double mediaAritmetica (double ... números) {
        double suma = 0.0;

        for (double num: números) {
            suma += num;
        }

        return suma / números.length;
    }
```

El atributo `números.length` contiene la cantidad de parámetros recibidos en números.

```
    public static void main (String args[]) {
        double media;

        media = mediaAritmetica (-10, 5, 0, 7, 20);
        System.out.println("La media es = " + media);
    }
}
```

Comentario: El método `mediaAritmetica` divide la suma de todos los valores por `números.length`. ¿Qué ocurre cuando se pasa una lista de parámetros vacía? En este caso, se divide por 0. Ahora bien, como el resultado es un `double`, en Java se asigna el valor `NaN` (Not a Number). Esto es porque suma contiene el valor 0.0 de la inicialización. El bucle `for` no se ejecuta, pues la lista es vacía, resultando en la división 0.0/0 que es indeterminada. (Nota: al dividir por 0.0 con reales en Java, también se puede obtener +∞, -∞ o `NaN`, según el numerador sea positivo, negativo o cero, respectivamente.)

Ejercicio 2.16:

Se desea calcular la distancia recorrida por un móvil. Se sabe que el móvil sigue movimientos rectilíneos entre una serie de puntos. Se pide escribir un método que reciba una serie de puntos y determine la distancia recorrida por el móvil. Para probar el método, escriba un programa que calcule la distancia recorrida entre los puntos (0,0), (2,4), (4,5), (4, 6), (3,4) y (0,0) (vuelta al origen). Pruebe también qué ocurre cuando se llama al método con un solo punto.

Planteamiento: Para el programa, se define una clase llamada `Trayectoria`, que contendrá el método pedido en el enunciado, que se llamará `recorrido`. La clase `Trayectoria` contendrá un método `main()` donde se generan primero los puntos que pide el enunciado y luego se llama al método `recorrido()` con dichos puntos, posteriormente con un solo punto y, por último, sin ningún punto.

Cuando un método recibe un número arbitrario de argumentos *del mismo tipo* se puede especificar con la notación "...", que significa que se espera un número variable de argumentos. Con un bucle `for` se pueden recorrer todos los puntos de la secuencia (la secuencia se puede tratar como si fuera un array, consulte el Capítulo 6). Para calcular el recorrido, se considera un punto cada vez. Entonces se calcula la distancia de un punto al anterior, añadiendo el resultado a una variable que acumula el recorrido total.

Parámetros: Una secuencia arbitraria de puntos, denotado como `Punto ... puntos`.

Valor de retorno: La suma de las distancias entre los puntos considerados en orden, como la distancia será un valor real, el valor devuelto será del tipo `double`.

Solución:

```
class Trayectoria {

    public static double recorrido (Punto ... puntos) {
        double dist = 0.0;

        Punto anterior = puntos[0];    // Como primer punto se pone el que está en la posición 0.

        for (Punto actual : puntos) {
            dist += actual.distancia(anterior);
            anterior = actual;    // Se calcula la distancia desde el punto actual al anterior y se acumula en dist. Luego se actualiza el anterior punto con el valor del actual.
        }

        return dist;
```

```
    }

    public static void main (String args[]) {
        Punto p1, p2, p3, p4, p5;

        p1 = new Punto (0, 0);
        p2 = new Punto (2, 4);
        p3 = new Punto (4, 5);
        p4 = new Punto (4, 6);
        p5 = new Punto (3, 4);

        System.out.println("Se ha recorrido " +
                           recorrido (p1, p2, p3, p4, p5, p1));

        System.out.println("Se ha recorrido " + recorrido (p1));

        System.out.println("Se ha recorrido " + recorrido ());
    }
}
```

Comentario: Hay varios puntos que resaltar en este ejercicio; nótese en primer lugar el uso de recorrido en el primer caso: el punto p1 se pasa dos veces, como origen y final de la trayectoria. Por otro lado, al ejecutar este programa, se imprime lo siguiente:

```
Se ha recorrido 14.94427190999916
Se ha recorrido 0.0
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 0
        at Trayectoria.recorrido(Trayectoria.java:7)
        at Trayectoria.main(Trayectoria.java:30)
```

La primera línea corresponde al recorrido entre los puntos. La segunda, corresponde a la llamada a `recorrido` con un solo punto. Es conveniente considerar si es correcto que al "recorrer" un solo punto se devuelva un cero, pues con un solo punto no ha habido trayectoria real. La tercera línea es inaceptable en la mayoría de los casos: se lanza una excepción (`ArrayIndexOutOfBoundsException`) por intentar acceder al primer elemento de la secuencia en la inicialización de (`Punto anterior = puntos[0];`) cuando en realidad no hay ningún punto. Se podría haber protegido el método comprobando el número de parámetros pasados (`puntos.size`), como en el siguiente código:

```
class Trayectoria {

public static double recorrido2 (Punto ... puntos) throws Exception {
        double dist = 0.0;

        if (puntos.length == 0)
          throw new Exception("No hay trayectoria");

        Punto anterior = puntos[0];

        for (Punto actual : puntos) {
          dist += actual.distancia (anterior);
          anterior = actual;
```

En el método `recorrido2` comprueba que se le pasan parámetros.

```
        }

        return dist;
    }

    public static void main (String args[]) throws Exception {
        Punto p1, p2, p3, p4, p5;

        p1 = new Punto (0, 0);
        p2 = new Punto (2, 4);
        p3 = new Punto (4, 5);
        p4 = new Punto (4, 6);
        p5 = new Punto (3, 4);

        System.out.println("Se ha recorrido " + recorrido2 (p1, p2, p3, p4, p5, p1));

        System.out.println("Se ha recorrido " + recorrido2 (p1));

        System.out.println("Se ha recorrido " + recorrido2 ());
    }

}
```

El método `main` delega excepciones, dado que se invoca al método `recorrido2` que las lanza. Consulte el Capítulo 4.

De esta forma, se sigue lanzando la excepción en el método `main()`, pero la responsabilidad ahora recae en el uso del método, no en la codificación del mismo. Esta diferencia es sutil, pero MUY importante, pues cada programador debe responsabilizarse de su código. En efecto, ahora al ejecutar el programa con este nuevo método,

```
Se ha recorrido 14.94427190999916
Se ha recorrido 0.0
Exception in thread "main" java.lang.Exception: No hay trayectoria
        at Trayectoria.recorrido2(Trayectoria.java:21)
        at Trayectoria.main(Trayectoria.java:46)
```

la excepción que se lanza es de tipo Exception, con el mensaje "`No hay trayectoria`", en contraposición del caso anterior que se lanzaba la excepción `ArrayIndexOutOfBoundsException`. En el primer caso, hay un error de uso, en el segundo un error de codificación: la responsabilidad está clara.

Aviso: En el método `recorrido` no se comprueba que no se pasen ningún punto al método o que alguno sea `null`. Tampoco se comprueba que no se pase ningún punto al método. Adicionalmente, si se pasa un solo punto, devuelve 0, lo que podría no tener mucho sentido. El caso de qué un punto fuese `null`, se podría codificar el bucle `for` para detectar el caso y lanzar la correspondiente excepción, como se muestra a continuación:

```
for (Punto actual : puntos) {
    if (actual == null)
        throw new Exception ("Hay un punto nulo");

    dist += actual.distancia(anterior);
    anterior = actual;
}
```

Por otra parte, aunque se ha puesto en el método main() que este delega Exception, esto nunca debería ocurrir, sólo se pone para simplificar el ejercicio, ya que no se tratan excepciones con todo detalle. Sobre el tratamiento de excepciones de forma apropiada consulte el Capítulo 4.

Ejercicio 2.17:

Escriba la clase Punto con dos métodos llamados distancia*. Uno de ellos calcula la distancia a otro punto y el otro calcula la distancia al origen.*

Planteamiento: La clase Punto, como en ejercicios anteriores, tendrá dos atributos para las coordenadas x e y. Para los métodos distancia, se proporciona uno que admite como parámetro un Punto y devuelve un double con la distancia a dicho punto; asimismo, se escribirá otro método también llamado distancia, pero sin parámetros, que calcule la distancia al origen.

Parámetros: Uno de los métodos distancia aceptará una referencia a la clase Punto y el otro no dispone de parámetros.

Valor de retorno: Como la distancia entre dos puntos será un número real, ambos métodos devolverán un valor del tipo double.

Solución:

```
import java.lang.Math;

public class Punto {

    private double x, y;

    public Punto (double x, double y) {
        this.x = x;
        this.y = y;
    }

    public double dameX(){
        return x;
    }

    public double dameY(){
        return y;
    }

    public double distancia() {
        return Math.sqrt(x * x + y * y);
    }

    public double distancia(Punto p) {   <-- Sobrecarga del método distancia.
        double diffX = x - p.x;
        double diffY = y - p.y;

        return Math.sqrt(diffX * diffX + diffY * diffY);
    }

}
```

Comentario: Cuando se encuentran varios métodos con el mismo nombre, pero que aceptan diferentes parámetros, bien por diferente cantidad de parámetros o bien por tener los parámetros diferente tipo, se dice que el método está sobrecargado. Para la sobrecarga no se tienen en cuenta ni las diferencias de acceso, ni el tipo devuelto ni las excepciones lanzadas. La sobrecarga de métodos es útil cuando se desean hacer cosas similares con diferentes parámetros.

Aviso: Usar la sobrecarga en exceso dificulta la legibilidad del programa y puede comprometer, por tanto, su mantenimiento.

Ejercicio 2.18:

Escriba una clase que represente un reloj que señale la hora, el minuto y el segundo. La clase dispondrá de dos constructores, uno sin parámetros que pone el reloj a 0:0:0 y otro al que se le pasa la hora, los minutos y los segundos.

Se proporcionarán los siguientes métodos:

- *Uno que da la hora, los minutos y los segundos, separados por el carácter ":", en una cadena.*
- *Otro que también da la hora pero en formato 24 horas (como el anterior) o en formato 12, en cuyo caso debe distinguir entre "am" (para las horas de 0 a 11) o "pm" (para las horas de 12 a 23), también en una cadena.*
- *Un método para poner el reloj en hora. Se le pasa la hora y los minutos, poniendo los segundos a 0.*
- *Un método para poner el reloj en hora al que, además, se le pasan los segundos.*

Planteamiento: Existen muchos problemas en los que la representación de la información admite diversos formatos (por ejemplo, coordenadas polares o cartesianas en el ejercicio de la clase Punto). En general, lo más adecuado es trabajar en un formato interno, que sea cómodo y fácil de programar y sólo las funciones que toman datos o los devuelven se preocupan del formato. En este caso, hay que elegir entre representar un reloj con las horas de 0 a 23 o bien de 0 a 12 y seleccionar "am" o "pm". El formato de 0 a 23 es más cómodo, pues no hay que guardar otro atributo con el uso horario.

Los atributos serán, por tanto, tres enteros, para la hora, los minutos y los segundos. No obstante habrá que comprobar que los valores son correctos, tanto en el constructor como en los métodos que ponen la hora. Si son incorrectos, se lanza una excepción descriptiva del problema. Como hay que hacer la comprobación en varios sitios, es conveniente escribir un método que lo haga. A este método se le llamará `compruebaHora`.

Existirán dos métodos que dan la hora, que se llamarán `dameHora()`; el primero, sin parámetros, la devolverá como un objeto `String`. El segundo necesita un parámetro para determinar el formato de la hora. Para ello, se proporciona un enumerado con los dos posibles formatos, el de 24 horas y el de 12. En el caso de que el formato sea de 12 horas y la hora sea de tarde (entre 12 y 23) habrá que restar 12 al valor de la hora y añadir la cadena "pm". En el mismo formato, pero con la hora entre 0 y 11 habrá que poner "am".

Por último, existen dos métodos para poner la hora, uno que toma dos parámetros, para la hora y los minutos, y otro que toma tres parámetros, la hora, los minutos y los segundos.

Parámetros: La hora, los minutos y los segundos serán representados en todos los casos con valores del tipo `int`. No obstante, se comprobarán los rangos adecuados. Se proporciona un enumerado para distinguir los formatos de 24 horas y el de 12 horas.

Valor de retorno: El valor devuelto por los métodos `dameHora()` será un objeto de la clase `String` con la hora en el formato adecuado.

Solución:

```
enum TipoHorario { H24, H12 }; ◄── Enumerado adicional para determinar
                                    el formato de la hora.

class Reloj {
```

```
    private int hora, minutos, segundos;

 private void compruebaHora (int hora, int minutos, int segundos)
                              throws Exception {
        if ( hora < 0 || hora > 23 )
            throw new Exception ("Error en hora");
        if ( minutos < 0 || minutos > 59 )
            throw new Exception ("Error en minutos");
        if ( segundos < 0 || segundos > 59 )
            throw new Exception ("Error en segundos");
    }

    public Reloj() {
        hora = minutos = segundos = 0;
 }

 public Reloj(int hora, int minutos, int segundos) throws Exception {
       compruebaHora(hora, minutos, segundos);

       this.hora     = hora;
       this.minutos  = minutos;
       this.segundos = segundos;
 }

    public Reloj(int hora, int minutos) throws Exception {
        this(hora, minutos, 0);
    }

    String dameHora() {
        return hora + ":" + minutos + ":" + segundos;
    }

    String dameHora(TipoHorario tipo) {
        String res;

        if ( tipo == TipoHorario.H12 && hora >= 12 )
            res = "" + (hora-12);
        else
            res = "" + hora;

        res += ":" + minutos + ":" + segundo + ( (hora < 12) ? "am" : "pm" );

        return res;

    }

    void ponHora(int hora, int minutos) throws Exception {
        ponHora(hora, minutos, 0);
    }

 void ponHora (int hora, int minutos, int segundos) throws Exception {
```

Método auxiliar que comprueba que los valores de hora, minutos y segundos son correctos, o lanza una excepción.

Si compruebaHora encuentra algún valor incorrecto, lanzará una excepción y el objeto no se construirá.

Este método sobrecargado delega en su homónimo, sin repetir código.

```
        compruebaHora(hora, minutos, segundos);

        this.hora     = hora;
        this.minutos  = minutos;
        this.segundos = segundos;
    }
}
```

De nuevo se invoca a compruebaHora para determinar si los valores son correctos. Si no lo son la hora no se modifica.

Comentario: En este enunciado, existen muchos puntos abiertos. Por ejemplo, los nombres de los métodos, el tipo de representación de la información, etc. Generalmente, el punto más delicado es elegir un buen modelo de datos, que permita programar de forma cómoda, sencilla y legible.

Respecto del enumerado TipoHorario, los valores que contienen no pueden ser { 24H, 12H }, como podría parecer natural, puesto que los identificadores de Java no pueden empezar por un dígito. Por ello, se antepone la hache delante.

Aviso: Nótese que el constructor sin parámetros NO delega en el otro constructor mediante this(0, 0, 0). Esto es así para evitar lanzar excepciones (o capturarlas como se verá en ejercicios posteriores del Capítulo 4). En este caso, se justifica el evitar usar el this por la sencillez del código que resulta. Hay que tener en cuenta que no tiene sentido que el constructor sin parámetros se declare que lanza excepciones, pues siempre debe ser correcto o no existir.

Por otro lado, el método compruebaHora es privado. Aunque los derechos de acceso se tratan en el siguiente capítulo, baste decir que de esta forma la clase Reloj contiene los métodos tal como indica el enunciado, pues compruebaHora no puede invocarse desde fuera de la clase.

Ejercicio 2.19:

Añada al ejercicio anterior un método para poner la hora especificando si es AM o es PM.

Planteamiento: Una vez más se sobrecarga el método ponHora(), en este caso se le añade un parámetro para determinar si la hora es AM o PM. En principio, se podría pensar en pasar una String, con los valores "am" o "pm". Pero ello implicaría comprobar que no se ponga otra cadena o un null. Es mucho mejor añadir un enumerado con los valores { AM, PM }, llamado TurnoHorario.

Al añadir el turno horario quiere decir que las horas se pasan en formato de 12 horas. Hay que comprobar, entonces, que el rango de la hora es de 0 a 11. Se puede optar por modificar compruebaHora, que comprobaba la hora en formato 24 horas o bien en añadir una línea de comprobación adicional. La solución mejor dependerá del uso posterior que se quiera dar a compruebaHora. En este caso, se añade una línea adicional.

Parámetros: Además de la hora, los minutos y los segundos como int, el turno de horario, de tipo TurnoHorario.

Valor de retorno: El método no necesita devolver ningún valor de retorno por lo que se utiliza la cláusula void.

Solución:

```
// Sólo se incluye el enumerado con el turno y el nuevo método ponHora
enum TurnoHorario { AM, PM };

void ponHora(int hora, int minutos, int segundos, TurnoHorario t) throws Exception {

    compruebaHora (hora, minutos, segundos);

    if ( hora > 11 ){
        throw new Exception ("Hora incorrecta en horario de 12 horas");
    }
```

```
    if ( t == TurnoHorario.PM ){
        hora += 12;
    }

    this.hora     = hora;
    this.minutos  = minutos;
    this.segundos = segundos;
}
```

Se pasa a formato de 24 horas.

Comentario: La hora se almacena en formato 24 horas, como se explicó en el ejercicio anterior. Por ello, si la hora es PM, se suman 12 horas al valor de hora recibido.

También se podría haber modificado el método `compruebaHora()`, quedando un código algo más limpio. El nuevo `compruebaHora()` y `ponHora()` quedarían de la siguiente forma:

```
private void compruebaHora (int hora, int minutos, int segundos, TipoHorario t)
                             throws Exception {
    if ( t == TipoHorario.H24){
        if ( hora < 0 || hora > 23 ){
            throw new Exception ("Error en hora formato 24H");
        }
    }

    if ( t == TipoHorario.H12){
        if ( hora < 0 || hora > 11 ){
            throw new Exception ("Error en hora formato 12H");
        }
    }

    if ( minutos < 0 || minutos > 59 ){
        throw new Exception ("Error en minutos");
    }

    if ( segundos < 0 || segundos > 59 ){
        throw new Exception ("Error en segundos");
    }
}
```

Comprobación del rango de la hora según el tipo de horario.

```
void ponHora (int hora, int minutos, int segundos, TurnoHorario t)
                           throws Exception {
    compruebaHora (hora, minutos, segundos, TipoHorario.H12);

    if ( t == TurnoHorario.PM ){
        hora += 12;
    }

    this.hora     = hora;
    this.minutos  = minutos;
    this.segundos = segundos;
}
```

Sigue siendo necesario pasar al formato interno del reloj de 24 horas.

Aviso: Evidentemente, si se introduce esta segunda codificación en la clase `Reloj`, habrá que modificar todas las invocaciones del método `compruebaHora`, especificándose si el formato de reloj es de 24 horas o 12 horas, según corresponda.

Ejercicio 2.20:

Se define la serie de Fibonacci como 1, 1, 2, 3, 5, 8, 13, 21,... donde los dos primeros términos son 1 y cada término restante es la suma de los dos anteriores. Formalmente, considerando el 0 como índice del primer elemento, el término a_i se calcula como:

$$a_i = \begin{cases} 1 & i = 0,\ i = 1 \\ a_{i-1} + a_{i-2} & i > 1 \end{cases}$$

Se pide escribir dos métodos que calculen el término i-ésimo de la serie de Fibonacci, uno de ellos de forma recursiva y otro de forma iterativa.

Planteamiento: Se llamará `fibR` al método recursivo y `fibI` al método iterativo. Ambos tomarán como parámetro el índice del término que se desea calcular. Fíjese que el primer elemento se denota como a_0, empezando con el índice 0. También ambos lanzarán una excepción cuando el índice sea negativo (fuera de rango).

Para la versión recursiva, se escribe la función prácticamente como se define formalmente: con un `if` se determina si el índice es 0 o 1, en cuyo caso se devuelve 1; en caso contrario, se devuelve la suma computando recursivamente (esto es, llamando a `fibR`) el valor de los anteriores términos. Esto se escribe como `fibR(i-1) + fibR(i-2)`.

Para la versión iterativa, hay que idear alguna forma de ir computando los términos hasta llegar al deseado. Para ello, se almacenarán los valores inmediatamente anteriores en dos variables, llamadas `último` y `penúltimo`. La suma de estos valores se almacenará en otra variable, llamada `res`. Con un bucle `for` se calculan los términos hasta el i-ésimo, de forma que en cada iteración, `res` se computa como `último + penúltimo` y luego se actualizan sus valores. Las tres variables se inicializan a 1 y el bucle no se ejecutará para los dos primeros términos.

Parámetros: El índice del término a calcular, como un valor del tipo `int`.

Valor de retorno: El cómputo de la función de Fibonacci, como `int`. Podría también devolverse un valor `long` si se desea poder calcular mayores valores de la función de Fibonacci.

Solución:

```
public int fibR(int i) throws Exception {
    if (i < 0){
        throw new Exception ("Índice negativo");
    }

    if ( i == 0 || i==1 ){
        return 1;
    }else{
        return fibR(i - 1) + fibR(i - 2);
    }
}
```

Condición de convergencia: las llamadas recursivas terminan alguna vez.

Llamada recursiva. Hay que asegurarse de que los valores que se pasan convergen a la condición de parada.

```
public int fibI(int i) throws Exception {
    if (i < 0){
        throw new Exception ("Índice negativo");
    }
```

```
        int último = 1, penúltimo = 1, res = 1;

        for (int n = 1; n < i; n++) {  <-- El bucle no se ejecuta para los términos 0 y 1.
            res = último + penúltimo;

            penúltimo = último;
            último    = res;  <-- Hay que tener cuidado en el orden de actualización.
        }

        return res;
    }
```

Comentario: No siempre será fácil encontrar una versión iterativa de un algoritmo o función definido recursivamente. Si es posible, como en este caso de Fibonacci, el algoritmo se denomina recursivo primitivo.

Aviso: Las funciones recursivas tienen una codificación mucho más legible y elegante de forma recursiva que iterativa. Por tanto, la versión recursiva es mucho más fácil de mantener y de depurar que la iterativa. No obstante, la implementación recursiva consume más recursos y generalmente tarda más en ejecutarse que la versión iterativa. Cuál debe elegirse dependerá sobremanera de los requisitos de ejecución de nuestro programa.

Ejercicio 2.21:

La función de Ackerman se define para valores enteros no negativos n y m como:

$$A(m,n) = \begin{cases} n+1 & m=0 \\ A(m-1,1) & n=0, m>0 \\ A(m-1, A(m,n-1)) & n>0, m>0 \end{cases}$$

Se pide codificar un método que calcule la función de Ackerman y un programa que imprima los valores de Ackerman (0,0), Ackerman(2,2) y Ackerman (6,6).

Planteamiento: La definición de la función de Ackerman es recursiva, por lo que se implementa un método recursivo, que se llamará Ackerman, que acepte dos enteros y calcule la función. Como Ackerman se define para números no negativos, se lanzará una excepción si alguno de los dos parámetros es menor que 0. Luego, según los valores de `m` y `n`, se calcula el valor según la definición.
Para obtener los valores de Ackerman que pide el enunciado, se escribe un `main()` en la misma clase, que calcula e imprime por pantalla los resultados.

Parámetros: Dos variables para los valores de `m` y `n` de tipo `int`.

Valor de retorno: Un entero (`int`) con el valor de la función de Ackerman. Podría también devolverse un valor `long` si se desea poder calcular mayores valores de la función de Ackerman.

Solución:

```
public class Ackerman {

    public static int Ackerman (int m, int n) throws Exception {
        if ( n < 0 || m < 0 ){
            throw new Exception ("Parámetros no válidos.");
        }

        if ( m == 0 ){
```

```
            return n + 1;
        }

        if ( n == 0 ){
            return Ackerman (m - 1, 1);
        }

        return Ackerman (m - 1, Ackerman (m, n - 1));
    }

    public static void main(String[] args) throws Exception {
        System.out.println("Ackerman(0,0)= " + Ackerman(0,0));
        System.out.println("Ackerman(2,2)= " + Ackerman(2,2));
        System.out.println("Ackerman(3,3)= " + Ackerman(3,3));
        System.out.println("Ackerman(6,6)= " + Ackerman(6,6));
    }
}
```

Cuando el cuerpo del `if` contiene un `return`, muchas veces no se pone `else` para los otros casos.

Comentario: La función de Ackerman es una función recursiva no primitiva, esto es, no es posible codificar un método iterativo que la implemente. El inconveniente que tiene Ackerman es que consume muchos recursos, aunque el valor de la función no sea muy elevado. Como ejercicio, se podría calcular el número de veces que se llama al método Ackerman. De hecho, la salida de este programa es:

```
Ackerman(0,0)= 1
Ackerman(2,2)= 7
Ackerman(3,3)= 61
Exception in thread "main" java.lang.StackOverflowError
        at Ackerman.Ackerman(Ackerman.java:20)
        at Ackerman.Ackerman(Ackerman.java:21)
        at Ackerman.Ackerman(Ackerman.java:21)
        at Ackerman.Ackerman(Ackerman.java:21)
        at Ackerman.Ackerman(Ackerman.java:21)
        at Ackerman.Ackerman(Ackerman.java:21)
        at Ackerman.Ackerman(Ackerman.java:21)
```

donde la última línea se repite un número muy grande de veces, que depende de la configuración del sistema, la cantidad de memoria instalada, etc.

Aviso: Existen algunos mecanismos, como la Programación Dinámica, para tratar algunos de estos problemas, pero el tratamiento de los mismos y su justificación excede del propósito de este libro. En el caso de la función de Ackerman, una implementación mediante Programación Dinámica es una solución muy adecuada. De todas formas, incluso para valores no muy grandes de m y n, el resultado de Ackerman es demasiado grande para almacenarlo en un entero.

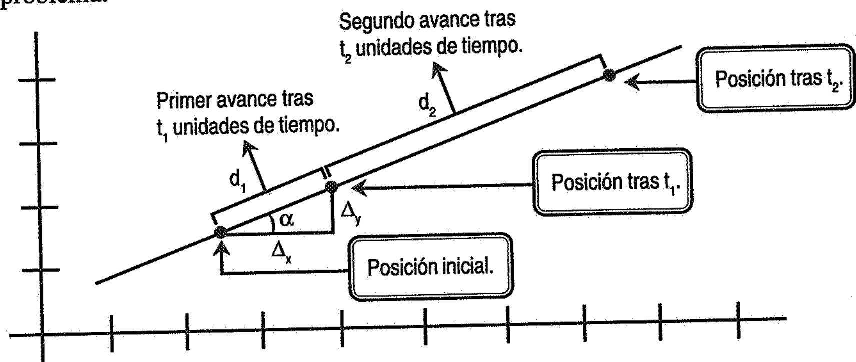
Ejercicio 2.22:

Escriba un programa que permita modelar el movimiento rectilíneo uniforme de un móvil en un plano. El móvil puede seguir una trayectoria en cualquier recta del plano. Se conoce la posición inicial del móvil, así como su velocidad lineal. Escriba, asimismo, un programa que ejercite dichas clases, de forma que se instancie un móvil que se mueve a una velocidad de 2.4 metros/segundo, que parte de la posición (2.1, 5.2) y se mueve sobre la recta y = 2x + 1. El móvil se irá moviendo en incrementos de tiempo arbitrarios. El programa ha de determinar las posiciones en las que se encuentra el móvil tras el paso de 3.5, 4, y 6.2 unidades de tiempo

respectivamente y las imprimirá por pantalla. (Nota: Evidentemente, el punto de partida del móvil ha de pertenecer a la recta que define el movimiento.)

Planteamiento: Este ejercicio tiene como objeto el identificar las clases más adecuadas para representar el problema, así como los atributos que contienen cada una de ellas y su interrelación (véase la sección "Comentarios", al final del ejercicio).

Lo primero es idear la solución al problema. Luego se modelan los conceptos encontrados en el problema y en la solución. Por último, se escribe un programa (`main()`) como pide el enunciado. La siguiente figura representa el problema:



El móvil comienza en una posición. Para saber cuál es la posición tras *t* unidades de tiempo, se calcula el recorrido en esas *t* unidades de tiempo a una velocidad *v* (*d= v * t*). Como el recorrido es lineal, el punto alcanzado dentro de la trayectoria se calcula sumando al origen los incrementos en abcisas y ordenadas, respectivamente *d*coseno(α)* y *d*seno(α)*, siendo α la pendiente de la recta.

Por tanto, partiendo de un punto, se calcula primero la distancia recorrida. Después se determinan los incrementos en abcisas y ordenadas. Luego se suman al punto en el que está situado el móvil, para obtener el punto destino. Tras cada uno de estos avances habrá de conservarse el punto anterior, para calcular el siguiente punto en la trayectoria.

Del enunciado y de la solución se deduce que el programa ha de manejar objetos que representen puntos, movimientos uniformes, rectas y móviles.

Primero se modela un punto, de forma similar a ejercicios anteriores en este mismo capítulo. Además, se le añade un método que devuelve una cadena con la representación del punto de forma "`(x, y)`".

Después se modela el movimiento uniforme, con una clase llamada `MovUniforme`, con un único atributo, la velocidad lineal del movimiento. También dispondrá de un constructor en el que se inicializa su atributo y un método que, dado un intervalo de tiempo (en unidades de tiempo), calcula el recorrido que se hace a dicha velocidad.

También hay que modelar lo que es una trayectoria rectilínea en un plano, es decir, una recta. Existen multitud de representaciones matemáticas de una recta, así que se elige una que sea cómoda. Por ejemplo, se usa la representación de una recta como $y = ax + b$, como propone el enunciado. En el constructor de la clase `Recta`, se le pasarán los valores de *a* y *b*. De la recta es necesario conocer el ángulo de la `pendiente`, por lo que se proporciona un método que la calcule, llamado pendiente, que devolverá un `double`.

Con estas tres clases, ahora se puede modelar un móvil, que será una clase llamada `Móvil`, que tendrá como parámetros la posición inicial del móvil, la posición actual del mismo, la posición anterior a la última vez que avanzó el móvil, el movimiento que ejecuta y la trayectoria que sigue. Dispondrá de un método que devuelva la posición actual y un método que permita avanzar *t* unidades de tiempo.

Con todo ello, el programa principal queda relativamente sencillo: Primero se crea un punto en la posición inicial del móvil (2.1, 5.2). Luego se crea una recta que representa la trayectoria del móvil ($y= 2x +1$). También se crea un movimiento uniforme con velocidad 2.4. El orden de creación de estos tres objetos es irrelevante. Pero con los tres, ya se puede crear un objeto de la clase `Móvil`, al que se llamará `m`. Entonces, alternativamente, se imprime la posición actual del móvil, y el móvil avanza según indica el enunciado: 3.2, 4.0 y 6.2 segundos. Entre cada avance, se imprime la posición actual.

Solución:

```
class Punto {
    private double x, y;

    Punto(double x, double y) {
        this.x = x;
        this.y = y;
    }

    double dameX(){
        return x;
    }

    double dameY(){
        return y;
    }

    String posición() {
        return "(" + x + "," + y + ")";
    }
}

class MovUniforme {
    private double vel;

    MovUniforme(double vel) {
        this.vel = vel;
    }

    double recorrido(double t) {
        return vel * t;
    }

}

class Recta {
    private double a, b;

    Recta(double a, double b){
        this.a = a;
        this.b = b;
    }

    double pendiente(){
        return Math.atan(a);
    }
}

class Móvil {

    private Punto posInicial;
```

```
    private Punto posAnterior;
    private Punto posActual;

    private Recta trayectoria;
    private MovUniforme mov;

    Móvil(Punto origen, Recta trayectoria, MovUniforme mov) {
        this.posAnterior = this.posActual=
        this.posInicial  = origen;

        this.trayectoria = trayectoria;
        this.mov = mov;
    }

    void avanza(double t) {
        Punto intermedio = new Punto (posActual.dameX(),
                                      posActual.dameY());

        double dist = mov.recorrido (t);

        double alfa = trayectoria.pendiente();

        double incrX = dist * Math.cos(alfa);
        double incrY = dist * Math.sin(alfa);

        posActual  = new Punto(posAnterior.dameX() + incrX,
                              posAnterior.dameY() + incrY);

        posAnterior = intermedio;
    }

    Punto posActual(){
        return posActual;
    }
}

public class problemaMóvil{

    public static void main(String[] args) {
        Punto inicial = new Punto (2.1, 5.2);

        Recta trayectoria = new Recta (2, 1);

        MovUniforme miMov = new MovUniforme (2.4);

        Móvil m = new Móvil(inicial, trayectoria, miMov);

        System.out.println("Me encuentro en " + m.posActual().posición());

        m.avanza(3.2);
        System.out.println("Me encuentro en " + m.posActual().posición());
```

avanza calcula la nueva posición pasados t segundos.

Punto intermedio para actualizar el punto anterior cuando acaben los cálculos.

Avanza e imprime la posición actual. Si el modelo de datos de nuestro problema es adecuado, el código resulta mucho más simple.

```
        m.avanza(4);
        System.out.println("Me encuentro en " + m.posActual().posición());

        m.avanza(6.2);
        System.out.println("Me encuentro en " + m.posActual().posición());
    }
}
```

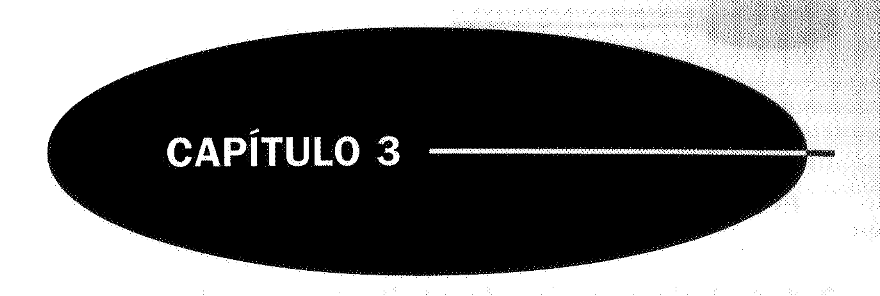
Comentario: La salida de este programa es:

```
Me encuentro en (2.1,5.2)
Me encuentro en (5.534600413439678,12.069200826879353)
Me encuentro en (6.393250516799597,13.786501033359919)
Me encuentro en (12.189138714479054,25.3782774289581)
```

El método `avanza` calcula la nueva posición pasados `t` segundos. Primero determina el recorrido lineal en ese plazo. Luego, según la pendiente de la recta que representa la trayectoria, calcula los incrementos en *x* e *y*. Por último, calcula la nueva posición actual a partir de la anterior. Y actualiza la anterior.

Aviso: Otra posibilidad es crear una única clase que tenga un montón de atributos para todos los valores que se han de almacenar o calcular. Esto, en general, es una mala idea.

Siempre hay que entender el problema antes de ponerse a codificar una posible solución. Luego, para cada concepto del problema o de la solución, se propone una clase que lo modela.



Ampliación de clases

3.1 ELEMENTOS DE CLASE (STATIC)

Los atributos y métodos de una clase precedidos por la palabra `static` se denominan elementos de clase. Los elementos de clase se comparten entre todas las instancias de la clase. Si se modifica un atributo de clase, todas las instancias de la clase ven dicha modificación.

Como ejemplo, si se declara un atributo de clase llamado `numAlumnos` y un método de clase llamado `imprimeTotalAlumnos`, el código quedaría así:

```
class Alumno {
    // Cuenta de alumnos matriculados,
    // inicialmente a 0
    static int numAlumnos = 0;

    // Resto de los atributos que se han omitido por simplificación

    // Se omiten las excepciones por claridad
    Alumno(String nombre, String apellidos, int año) {
        // Se Incrementa el núm. de alumnos matriculados
        numAlumnos++;

        // Resto del código del constructor que se ha omitido por simplificación
    }

    static void imprimeTotalAlumnos() {
        System.out.println("Número total de matriculados " + numAlumnos);
    }
}
```

Dado que tanto el atributo `numAlumnos` como el método `imprimeTotalAlumnos` existen aunque no haya objetos se les llamará desde el programa principal de la siguiente forma:

```
    public static void main(String args[]) {
```

```
        Alumno.imprimeTotalAlumnos();
    }
```

Es decir, con el nombre de la clase y el nombre del método.

3.1.1 Valor inicial de atributos de clase

Como los atributos y métodos estáticos se pueden utilizar aunque no exista ningún objeto de la clase, deben tener siempre un valor correcto. Los atributos se pueden declarar e inicializar de la misma forma que las variables normales.

Si los valores iniciales de los atributos de clase requieren cálculos adicionales iniciales, se usa un bloque de inicialización de atributos estáticos, de la siguiente forma:

```
class Alumno {
    static int numAlumnos;

    static {
        numAlumnos = 0;
    }

    ...
}
```

⇨ Sobre el uso de elementos de clase, consulte los Ejercicios del 3.1 al 3.8.

3.2 DERECHOS DE ACCESO

El estado de un objeto es el conjunto de los valores de sus atributos. Una modificación arbitraria, intencionada o por error, de este estado puede dar lugar a inconsistencias o comportamientos indeseados del objeto. Sería deseable poder controlar el acceso a los atributos de los objetos.

Java proporciona mecanismos de acceso a los componentes de una clase, de forma que es posible ajustarlo a las necesidades de los objetos. Para ello, se antepone a la declaración el modificador de acceso que se requiere:

- **Acceso privado** (`private`): Los elementos privados sólo se pueden usar dentro de la clase que los define, nunca desde ninguna otra clase.
- **Acceso de paquete**: No se pone nada. El acceso a estos componentes es libre dentro del paquete en el que se define la clase.
- **Acceso protegido** (`protected`): Los elementos protegidos sólo se pueden usar dentro de la clase que los define, aquellas clases que la extiendan y cualquier clase en el mismo paquete.
- **Acceso público** (`public`): Dicho elemento se puede usar libremente.

Así, para limitar el acceso a los atributos `nombre`, `apellidos` y `añoDeNacimiento` de un objeto alumno se declararían como:

```
enum Horario { MAÑANA, TARDE }

class Alumno {
    private String nombre;
    private String apellidos;
    private int añoDeNacimiento;
```

```
        private int NP; // Número Personal: identificativo único
        private String grupo;
        private Horario horario;

        // Resto de la clase Alumno
    }
```

De esta forma, el intento de cambiar cualquier atributo privado dará lugar a un error de compilación, y sólo se podrá acceder a la parte pública de la clase. Sólo se pueden modificar los valores de los atributos desde los propios métodos de la clase. De esta forma, aunque se puedan modificar los atributos a través de un método `ponGrupo()`, éste puede verificar que la modificación es segura y en caso contrario lanzar una excepción.

3.3 PAQUETES

Los paquetes son agrupaciones de clases, interfaces y otros paquetes (subpaquetes), normalmente relacionados entre sí. Los paquetes proporcionan un mecanismo de encapsulación de mayor nivel que las clases. Los paquetes permiten unificar un conjunto de clases e interfaces relacionados funcionalmente. Por ejemplo, el paquete `java` engloba una serie de paquetes con utilidades de soporte al desarrollo y ejecución de la aplicación. Contiene, a su vez, los subpaquetes `util` o `lang`. Para indicar que la clase que se está escribiendo pertenece a un paquete, la primera sentencia debe tener la sintaxis:

```
package nombrePaquete;
```

Todos los elementos contenidos en el fichero en el que aparece tal declaración formarán parte del paquete `nombrePaquete`.

3.3.1 Uso

Cuando en una clase se quieren utilizar componentes que están en otro paquete diferente, se añade una declaración de importación, que puede tener las siguientes formas:

```
// Importación de la clase "Alumno"
// del paquete "Matricula"
import Matricula.Alumno; // importación de la clase "Alumno" del paquete "Matricula"
import Matricula.*; // importación del paquete "Matricula", incluye la clase Alumno
```

3.3.2 Nombres

El nombre del paquete, como todos los identificadores, debe ser representativo de la agrupación que contiene. El nombre puede contener la declaración de subpaquete. Incluso se puede recomendar incluir el dominio de la empresa para que quede identificado de forma única respecto a otros paquetes que pudieran existir o comprarse a otros proveedores.

```
package com.empresa.Matricula;
```

Los derechos de paquete no se modifican porque estén contenidos en otro paquete. Un paquete que contenga dos subpaquetes no implica permisos de acceso entre ellos. Es decir, agrupar paquetes es conveniente para el posterior desarrollo de código, pero no para modificar los derechos de acceso.

⇨ Sobre los paquetes consulte el Ejercicio 3.9.

3.4 CLASES INTERNAS

Una *clase interna* es una clase cuya definición está dentro de otra clase. Una clase exterior se denomina clase externa o clase contenedora. Las clases internas permiten evitar la proliferación de clases muy pequeñas que en muchos casos sólo se usan dentro de una sola clase. El uso más común es la creación de adaptadores y manejadores de eventos, como se verá en los últimos capítulos, así como en la implementación de interfaces tipo `Iterator`. Otros usos quedan fuera del ámbito de este libro.

Hay distintos tipos de clases internas pero, por simplicidad, sólo se presentan las clases internas que se definen como miembros normales no estáticos de una clase contenedora. Como son miembros de la clase contenedora sólo puede existir un objeto de la clase interna cuando exista un objeto de la clase contenedora.

Las clases internas pueden acceder directamente a todos los miembros de la clase contenedora. Y ello es independiente de los modificadores de control de acceso que tengan los miembros de modo que, por ejemplo, puede utilizar directamente una variable miembro aunque esté declarada como privada.

Las clases internas no estáticas tienen algunas limitaciones: los nombres de los miembros de la clase contenedora tienen que ser diferentes de los de la clase interna, una clase interna no puede contener ningún miembro estático y no es posible crear un objeto de la clase interna sin tener un objeto de la clase contenedora.

En el siguiente ejemplo se añade una clase interna a la clase `Alumno` previamente definida para poder tener los datos sobre la dirección del alumno. Se ha simplificado la dirección a sólo la calle y el número. Se han añadido los métodos `toString()` tanto en `Alumno` como en `Dirección` que devuelve el contenido del objeto en forma de cadena y permite la escritura directa por pantalla de un objeto utilizando el método `System.out.println()`.

```
public class Alumno {

    private String nombre;
    private String apellidos;
    private int añoNacimiento;
    private Horario horario;
    private Direccion direccion;

    public Alumno(String nombre,
                  String apellidos,
                  int año,
                  Horario horario,
                  String calle,
                  int num) {
        this.nombre    = nombre;
        this.apellidos = apellidos;
        añoNacimiento = año;
        this.horario   = horario;
        direccion     = new Direccion(calle, num);
    }

    public void ponDireccion(Direccion dir) {
        direccion= dir;
    }

    // devuelve los datos del objeto en formato cadena
    public String toString(){
        return nombre + " " + apellidos + " "
               + añoNacimiento + " "
               + horario + " " + direccion;
```

```
        }

        class Direccion{
            String calle;
            int numero;

            Direccion(String calle, int numero){
                this.calle  = calle;
                this.numero = numero;
            }

            public String toString() {
                return calle + " " + numero;
            }
        }

        public static void main(String args[]) {
            Alumno alumno = new Alumno("Balta", "Fernández", 1991,
                                       Horario.MAÑANA, "Zamora", 23);
            System.out.println("Datos alumno: " + alumno);
            Alumno.Direccion direccion = alumno.new Direccion("Figueruela", 28);
            alumno.ponDireccion(direccion);
            System.out.println("Datos alumno: " + alumno);
        }
    }
```

Con las clases internas se introduce una nueva sintaxis en el nombrado de las clases, en el operador `new`, de creación de una instancia y en la autorreferencia `this`. Como se puede ver en el ejemplo, si se quiere utilizar directamente la clase interna, por ejemplo, para crear un nuevo objeto, hay que hacerlo `NombreClaseContenedora.NombreClaseInterna`. Si la clase interna no se ha declarado privada se puede crear un objeto de esa clase interna mediante `referenciaObjetoClaseContenedora.new NombreClaseInterna()`.

Si dentro de la clase interna se desea hacer una referencia al objeto actual de la clase contenedora se hace mediante `NombreClaseContenedora.this`.

Java también permite crear clases locales. Las *clases locales* son aquellas que se definen dentro de un método y que, por tanto, sólo son visibles y utilizables en dicho método. Así mismo, permite crear *clases anónimas* que son clases locales sin nombre. Las clases locales y las clases anónimas se usan de forma habitual en el trabajo con archivos y en la creación de interfaces gráficas de usuario de las que se verán ejemplos en el Capítulo 10.

➪ Sobre la declaración y uso de clases internas consulte los ejercicios del Capítulo 10.

3.5 IMPORTACIÓN ESTÁTICA DE CLASES

Para acceder a los miembros estáticos de una clase hay que escribir el nombre de la clase, un punto y el nombre del atributo o método. Esta sintaxis puede comprometer la legibilidad de un programa, sobre todo cuando aparecen muchos métodos y atributos estáticos con el nombre de la clase. Java proporciona un mecanismo para la importación de miembros estáticos, anteponiendo el cualificador `static` tras la importación. Se permite tanto la importación individual:

```
import static java.lang.Math.PI;
```

como todos los miembros de la clase:

```
import static java.lang.Math.*;
```

De esta forma, se puede escribir un programa para el cálculo genérico de un logaritmo como:

```
import static java.lang.Math.*;

class LogGenerico {
    static double x = ...; // poner valor double
    static double a = ...; // poner valor double

    // calcula logaritmo de "x" en base "a"
    public static void main (String args[]) {
        double res;
        res = log(x) / log(a); // uso de log sin cualificar
        System.out.println("res= " + res);
    }
}
```

En este ejemplo el método `log()` pertenece a la clase `Math`.

⇨ Sobre la importación estática de clases consulte el Ejercicio 3.21.

3.6 CLASES PREDEFINIDAS

Uno de los puntos fuertes de Java es la gran cantidad de clases predefinidas que aporta. Abarcan temas como comunicaciones, web, diseño gráfico, utilidades matemáticas, contenedores genéricos y muchas más. En este apartado se verán algunas de las más utilizadas:

- Envoltorios de los tipos simples.
- `String`, para la manipulación de cadenas de texto.
- `Math`, biblioteca de funciones matemáticas.

3.6.1 Envoltorios

Los tipos predefinidos en el lenguaje Java son tipos simples, en el sentido de que no son clases. Por conveniencia, existen unas clases predefinidas, denominadas envoltorios (*wrappers*, en inglés), para estos tipos simples. Estas clases envoltorio proporcionan métodos de clase, útiles para convertir cadenas de texto al tipo adecuado, imprimir con varios formatos, constantes estáticas con el valor mínimo y máximo del tipo, etc. Además, estas clases envoltorio generan automáticamente una instancia cuando se usan tipos simples en contextos en los que se espera un objeto. Así mismo, los envoltorios son compatibles con los tipos simples que representan, por lo que pueden usarse en expresiones en que se espera el tipo simple. Los envoltorios definidos en Java pueden verse en la Tabla 3.1.

Los tipos `byte` y `short` se incluyen en el lenguaje Java por razones de eficiencia, por lo que no tienen envoltorios predefinidos, utilizándose la clase `Integer`.

Todos los envoltorios disponen, entre otros, de los siguientes constructores y métodos:

- Constructor a partir de un valor del tipo simple,

```
Character miLetra  = new Character('H');
Integer   miEntero = new Integer(1024);
```

Tabla 3.1. Envoltorios definidos en Java.

Envoltorio	Tipo predefinido
`Boolean`	`boolean`
`Character`	`char`
`Integer`	`int`
`Long`	`long`
`Float`	`float`
`Double`	`double`

- Constructor que toma una cadena y lo traduce al tipo correspondiente,

```
Float miRealCorto = new Float("12.3E-2");
```

- Método `toString()` que transforma el valor en una cadena,

```
System.out.println("Mi entero es " + miEntero); //llamada implícita a toString()
```

- Método que devuelve el valor primitivo. El nombre del método es la concatenación del tipo simple con la palabra Value, es decir, `charValue()` en el envoltorio `Character`, `intValue()` en el envoltorio `Integer`, etc.

```
int m = miEntero.intValue();
```

- Método `equals()` para comparar el valor entre envoltorios,

```
Integer otroEntero = new Integer(2048);
boolean res = miEntero.equals(otroEntero);
```

En el siguiente ejemplo se muestra la posibilidad de mezclar tipos simples y envoltorios en expresiones matemáticas realizándose la conversión entre ellos de forma automática.

```
public static void main(String args[]) {
    Integer envX = new Integer(7);
    int y = 8;
    double z;

    envX += y; // añade y a envX
    envX++; // incrementa en uno el valor de envX
    z = Math.sqrt(envX); // raíz cuadrada de envX, z = 4.0
}
```

Esta conversión automática permite utilizar un tipo primitivo en el ámbito en que se requiere un objeto, por ejemplo para añadir un entero a una lista de la clase ArrayList, como se verá en ejercicios del Capítulo 6.

Boolean
La única particularidad de `Boolean` es que al convertir una cadena, se traduce la cadena "`true`" (cualquier combinación de minúsculas y mayúsculas) al valor `true`. Cualquier otra cadena se traduce al valor `false`.

Character
El envoltorio `Character` proporciona métodos para determinar si un carácter es una letra o un dígito, espacio en blanco (' ', '`\t`', '`\n`', '`\f`' o '`\r`'), pasar de mayúsculas a minúsculas, etc. Existen algunas particularidades en estos

métodos dependiendo del alfabeto al que pertenecen los caracteres. Java, como utiliza Unicode para codificar los caracteres, permite otros alfabetos como el cirílico, griego, kanji, etc., cuyo comportamiento con las mayúsculas y minúsculas puede diferir de los alfabetos occidentales derivados del latín. Asimismo, los dígitos para dichos alfabetos también son diferentes de '0', '1', etc. Para estos detalles, consúltese un manual de referencia de Java.

Integer
Además de almacenar un valor int, se suele usar para los tipos predefinidos byte y short, ya que no tienen envoltorios propios.

Float y Double
En Java, los tipos float y double pueden contener el valor +∞ y el -∞, por ejemplo, en las divisiones por 0. Pero también algunas expresiones pueden dar lugar a valores que no representan un número, como por ejemplo Math.sqrt(-1). Esto en Java se representa con el valor NaN. Asimismo, existen métodos para determinar si un número es +∞, -∞ o NaN.

```
float s= 0;
s = 10 / s;
Float infinito= new Float(s);

// Imprimirá "infinito es Infinity"
System.out.println("infinito es " + infinito);
Float noNumero= new Float(Math.sqrt(-1));
// Imprimirá "noNumero es NaN"
System.out.println("noNumero es " + noNumero);
```

3.6.2 Math

La clase Math contiene constantes y métodos de uso común en matemáticas. Todas las operaciones que se llevan a cabo en dicha clase se realizan con tipo double. Contiene las constantes π (Math.PI) y el número de Euler, e (Math.E), ambos de tipo double. En las funciones trigonométricas, los ángulos están en radianes y los métodos devuelven valores double.

3.6.3 String

La clase String se usa para manejar cadenas de caracteres de cualquier longitud finita. Es una clase especial con soporte específico incluido en el lenguaje Java. Para crear un objeto String no hace falta llamar al constructor, basta escribirlo como un valor.

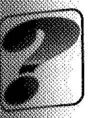
```
String nombre; // crea variable "nombre" de tipo String
String saludo  = "hola"; // crea un objeto String y lo asigna a la variable "saludo"
String mensaje = saludo;
```

La clase String tiene un tratamiento particular en Java, pues aparte de la construcción de objetos a partir de literales entre comillas, se pueden aplicar los operadores + y += que se usan para concatenar String.

```
String s = "Hola " + "Pedro."; // s valdrá "Hola Pedro."
s += " Hola " + "Juan."; // s valdrá "Hola Pedro. Hola Juan."
```

Para comparar si los valores de dos String son iguales se utiliza el método equals().

⇨ Sobre el uso de las clases predefinidas consulte los Ejercicios 3.10 al 3.20.



Problemas resueltos

ELEMENTOS DE CLASE (STATIC)

Ejercicio 3.1:

Escriba un programa que imprima en la pantalla las constantes de la clase Math.

Planteamiento: Para utilizar una constante de clase, se pone el nombre de la clase, un punto y el nombre de la constante. Las constantes son atributos calificados como final, esto es, su valor no puede cambiar. La clase `Math` proporciona dos constantes, el valor de π y el de la constante de Euler, llamados `Math.PI` y `Math.E`.

Solución:

```
import java.lang.Math;

public class PruebaCtesMath {

    public static void main(String args[]) {
        System.out.println("Pi vale " + Math.PI);
        System.out.println("E (cte de Euler) vale " + Math.E);
    }
}
```

Uso de la constante π, (`Math.PI`).

Comentario: La salida del programa es:

```
Pi vale 3.141592653589793
E (cte de Euler) vale 2.718281828459045
```

Ejercicio 3.2:

Escriba una clase de nombre Constantes que declare las siguientes constantes:

- *Velocidad de la luz, c : 2,9979 · 10^8 m/s*
- *Constante Universal de Gravitación, G: 6,67·10^{-11} N m^2/kg^2*
- *Constante de Planck, h: 6,6262 · 10^{-34} J·s*

Escribir asimismo un programa llamado pruebaConstantes, que cree un objeto de la clase Constantes y luego imprime los valores de las constantes y sus dimensiones.

Planteamiento: Una constante es un atributo al que se le ha declarado como `final`. Estos atributos sólo pueden tomar valor una vez, normalmente en la misma línea de la declaración. Como todos estos valores tienen rangos muy pequeños o grandes, deberán ser `double`.

Una constante es un atributo final, que no se puede modificar.

Solución:

```
class Constantes {
    public final double C = 2.9979E08;  // m/s VELOCIDAD DE LA LUZ
    public final double G = 6.67E-11;   // N m²/kg² CONSTANTE GRAVITARORIA UNIVERSAL (O DE NEWTON)
    public final double h = 6.6262E-34; // J•s CONSTANTE DE PLANK
}

class PruebaConstantes {
```

```
    public static void main (String args[]) {
        Constantes ctes = new Constantes ();

        System.out.println("La velocidad de la luz es " + ctes.C +
                           " en metros por segundo");
        System.out.println("La constante de gravitación universal es " + ctes.G +
                           " en Newtows por metros al cuadrado partido kilogramos al cuadrado");
        System.out.println ("La constante de Plank es " + ctes.h +
                            " en julios por segundo");
    }
}
```

Comentario: La salida del programa es:

```
La velocidad de la luz es 2.9979E8 en metros por segundo
La constante de gravitación universal es 6.67E-11 en Newtows por metros al cuadrado partido kilogramos
al cuadrado
La masa en reposo del electrón es 9.1096E-31 en kilogramos
La carga del electrón es 1.6022E-19 en culombios
La constante de Plank es 6.6262E-34 en julios por segundo
```

Aviso: Este programa adolece de un defecto: para acceder a las constantes hay que crear primero un objeto. Es mejor declararlas como constantes de clase, como se vio en el Ejercicio 3.1, en el que las constantes de la clase `Math` están disponibles sin necesidad de objetos. (Nota: además, `Math` está escrito de forma que no se pueden crear objetos.)

Ejercicio 3.3:

Escriba un programa para representar el consumo de energía de una instalación eléctrica. Para ello, se dispondrá de una clase que representa los aparatos conectados en la instalación. Cada aparato tiene un consumo eléctrico determinado. Al encender un aparato eléctrico, el consumo de energía se incrementa en la potencia de dicho aparato. Al apagarlo, se decrementa el consumo. Inicialmente, los aparatos están todos apagados. Además, se desea consultar el consumo total de la instalación.

Hacer un programa que declare tres aparatos eléctricos, una bombilla de 100 watios, un radiador de 2000 watios y una plancha de 1200 watios. El programa imprimirá el consumo nada más crear los objetos. Posteriormente, se enciende la bombilla y la plancha, y el programa imprime el consumo. Luego se apaga la plancha y se enciende el radiador y se vuelve a imprimir el consumo.

Planteamiento: Según el enunciado, la clase que modela a los aparatos eléctricos tiene dos datos, su potencia y si están encendidos o no. Esta clase se llamará `AparatoEléctrico`. Por otro lado, el consumo total debe ser un atributo accesible a todos los aparatos eléctricos, lo que se consigue con un elemento de clase. Para ello, se declara el atributo `consumoTotal`, de tipo `double` y se califica como `static`. Así, todas las instancias de `AparatoEléctrico` comparten el mismo atributo.

Para consultar el consumo total de la instalación, se proporciona un método que devuelva el valor de `consumoTotal`. Este método ha de ser también de clase (calificado como `static`) y se llamará `consumo()`.

Cuando un aparato eléctrico se enciende puede incrementar el `consumoTotal` con la potencia que consume. Esto lo llevará a cabo el método `enciende()`. Correspondientemente, cuando el aparato se apaga, se resta su potencia al consumo total. Y de nuevo, sólo si estaba encendido. Esto se implementa en el método `apaga()`. Evidentemente, hay que incrementar el consumo sólo si el aparato estaba encendido

Solución:

```
class AparatoEléctrico {
```

```
    static double consumoTotal = 0;

    static double consumo() {
        return consumoEléctrico;
    }

    private double potencia;
    private boolean encendido;

    AparatoEléctrico (double potencia) {
        this.potencia = potencia;
        encendido = false;
    }

    void enciende() {
        if ( ! encendido ){
            encendido = true;
            consumoEléctrico += potencia;
        }
    }

    void apaga() {
        if ( encendido ){
            encendido = false;
            consumoEléctrico -= potencia;
        }
    }
}

class ConsumoEléctrico {

    public static void main(String[] args) {
        AparatoEléctrico bombilla = new AparatoEléctrico (100);
        AparatoEléctrico radiador = new AparatoEléctrico (2000);
        AparatoEléctrico plancha  = new AparatoEléctrico (1200);

        System.out.println("El consumo eléctrico es " +
                           AparatoEléctrico.consumo());

        bombilla.enciende();
        plancha.enciende();
        System.out.println("El consumo eléctrico es " +
                           AparatoEléctrico.consumo ());

        plancha.apaga();
        radiador.enciende();
        System.out.println("El consumo eléctrico es " +
                           AparatoEléctrico.consumo ());
    }
}
```

Método de clase para acceder a un atributo de clase.

Al encender o apagar, sólo se modifica el consumoTotal si se cambia de estado.

Comentario: La salida por pantalla que genera este programa es:

```
El consumo eléctrico es 0.0
El consumo eléctrico es 1300.0
El consumo eléctrico es 2100.0
```

Aviso: El punto más delicado es olvidar que si se invoca el método `enciende()` dos veces seguidas, la segunda no debe incrementar el consumo. Podría ser muy incómodo lanzar una excepción en ese caso. Por ello, el método `enciende()` sólo incrementa el consumo si el aparato estaba apagado. Correspondientemente, sólo se decrementa el consumo si el aparato estaba encendido.

Ejercicio 3.4:

Añada a la clase Punto una constante llamada ORIGEN que sea el origen de coordenadas. La constante debe ser accesible de forma estática. Escriba un programa, de nombre PruebaPunto, que determine la distancia de los puntos (3,4), (0,4) y (2, -1) al origen usando la constante ORIGEN.

Planteamiento: Una constante es un atributo al que no se puede cambiar el valor una vez inicializado. Normalmente, son atributos públicos de clase. En este caso, el valor constante es un punto situado en (0, 0). Para declarar la constante se pone la palabra `final` delante de un atributo. El atributo no podrá cambiar de valor. Como se desea que la constante esté disponible de forma estática, se antepone también la palabra `static`. Así, se podrá usar la constante escribiendo `Punto.ORIGEN`, sin necesidad de tener objetos de la clase `Punto`.

Escribe: Se escribe el mensaje `"Distancia de p al origen"` seguido del valor.

Solución:

```
class Punto {

    public static final Punto ORIGEN = new Punto (0, 0);

    private double x, y;

    public Punto (double x, double y) {
        this.x = x;
        this.y = y;
    }

    public double dameX(){
        return x;
    }

    public double dameY(){
        return y;
    }

    public double distancia (Punto p) {
        double diffX = x - p.x;
        double diffY = y - p.y;

        return Math.sqrt(diffX * diffX + diffY * diffY);
    }
}
```

Las constantes sólo pueden tomar valor una vez. Se recomienda escribir su nombre en mayúsculas.

```
class PruebaPunto {
    public static void main(String[] args) {
        Punto p1 = new Punto (3, 4);
        Punto p2 = new Punto (0, 4);
        Punto p3 = new Punto (2, -1);

        System.out.println("Distancia de p1 al ORIGEN= " + p1.distancia(Punto.ORIGEN));
        System.out.println("Distancia de p2 al ORIGEN= " + p2.distancia(Punto.ORIGEN));
        System.out.println("Distancia de p3 al origen= " + p3.distancia(Punto.ORIGEN));
    }
}
```

Comentario: Es muy común declarar e inicializar las constantes simultáneamente. También es posible añadir una sección donde se inicializan los miembros estáticos de una clase. Algunas clases de utilidad pública tienen todos sus atributos y métodos declarados `static`, para usarlos directamente sin crear objetos. Un caso de ejemplo ya visto es la clase `Math`. La salida del programa es:

```
Distancia de p1 al ORIGEN= 5.0
Distancia de p2 al ORIGEN= 4.0
Distancia de p3 al origen= 2.23606797749979
```

Aviso: Puntos delicados acerca del problema o de la solución.

Ejercicio 3.5:

Escriba una clase llamada Elemento, que disponga de un atributo con su nombre. La clase debe contener un método llamado númeroDeElementos que devuelve el número total de elementos que se han instanciado.

Planteamiento: Para determinar el número total de elementos, se crea un atributo de clase que se incrementa cada vez que se crea una nueva instancia. Esto se hace en el constructor de `Elemento`. Además, se añade el método de clase `númeroDeElementos` que lo devuelve.

Solución:

```
class Elemento {
    private String nombre;

    Elemento(String nombre) {
        this.nombre = nombre;

        numElementos++;
    }

    static private int numElementos = 0;

    static int númeroDeElementos(){
        return numElementos;
    }
}

public class PruebaElemento {

    public static void main(String[] args) {
```

Además de la construcción habitual de un objeto de la clase, se actualiza el atributo de clase `numElementos`.

```
        Elemento H  = new Elemento("Hidrógeno");
        Elemento He = new Elemento("Helio");
        Elemento Li = new Elemento("Litio");

        System.out.println("Creados: " + Elemento.númeroDeElementos());
    }
}
```

Comentario: Este programa imprime "Creados: 3" por pantalla.

Ejercicio 3.6:

Escriba una clase de nombre Ítem, que acepta un nombre en su constructor. Cada ítem debe disponer de un número que servirá como identificador único de cada objeto. La clase Ítem dispondrá de un método para obtener el identificador y otro para obtener el nombre. Haga un programa de prueba que genere tres ítems, "uno", "dos" y "tres" y luego escriba los nombres e identificadores de cada ítem.

Planteamiento: La parte del atributo no presenta ningún problema. Para asignar un identificador único a cada instancia de la clase, se proporciona un atributo de clase, que no es más que un contador de objetos construidos. Además, se guarda el valor de ese contador en un atributo de *cada* objeto. Así, cada objeto recibe un número distinto, único. Luego, el programa de prueba consiste en crear los tres objetos Ítem con nombres. Es al acceder a los identificadores cuando se ve que cada uno ha recibido un número distinto.

Solución:

```
class Ítem {
    private static int contador = 0;

    private int Id;
    private String nombre;

    Ítem (String nombre) {
        this.nombre = nombre;

        Id = ++contador;   <-- Usamos un atributo de clase para
                               identificar a cada objeto de la clase.
    }

    int Id(){
        return Id;
    }

    String nombre(){
        return nombre;
    }
}

public class PruebaÍtem {
    public static void main(String args[]) {
        Ítem i1 = new Ítem("uno");
        Ítem i2 = new Ítem("dos");
        Ítem i3 = new Ítem("tres");

        System.out.println("Ítem " + i1.nombre() + " id " + i1.Id());
```

```
        System.out.println("Ítem " + i2.nombre() + " id " + i2.Id());
        System.out.println("Ítem " + i3.nombre() + " id " + i3.Id());
    }
}
```

Comentario: La salida del programa es:

```
Ítem uno id 1
Ítem dos id 2
Ítem tres id 3
```

Aviso: Como mecanismo de identificador de objetos, el sistema puede ser un poco pobre. Cuando los objetos desaparecen, quedan números sin asignar que no se volverán a usar. Todos los objetos en Java reciben un identificador único, pero basado en funciones de dispersión (hash function).

Ejercicio 3.7:

Se desea representar las bombillas que pueda haber en una casa. Cada bombilla tiene asociada un interruptor y sólo uno. Así mismo, existe un interruptor general de la casa. Un interruptor tiene dos estados, ON y OFF. Una bombilla luce si el interruptor general de la casa está ON y su interruptor asociado también. Escriba una clase de nombre Bombilla que permita modelar la información anterior. Para ello, la clase dispondrá de:

- *un método para cambiar el estado del interruptor de la bombilla*
- *un método para cambiar el estado del interruptor general de la casa*
- *un método que determina si una bombilla está luciendo o no*

Planteamiento: Los interruptores se representan por los estados ON y OFF. Para ello, se usa un enumerado con ambos valores, llamado `Estado`. Por otro lado, del enunciado se puede deducir que cada `Bombilla` tendrá un atributo que será su interruptor asociado. Ahora bien, el interruptor general es único y común para todas las bombillas. Esto se modela como un atributo de clase (`static`) que comparten todos los objetos. Asimismo, el método que pulsa dicho interruptor ha de ser de clase también. Por último, para el método que determina si la bombilla luce o no, hay que tener en cuenta que tanto el interruptor de la bombilla como el interruptor general deben estar a ON. Cuando uno de ellos esta a OFF la bombilla no luce.

Solución:

```
enum Estado { ON, OFF };

class Bombilla {
    // Parte estática que representa al Interruptor General
    private static Estado interruptorGeneral = Estado.OFF;

    public static void pulsaInterruptorGeneral(){
        if (interruptorGeneral == Estado.ON){
            interruptorGeneral = Estado.OFF;
        }else{
            interruptorGeneral = Estado.ON;
        }
    }

    // Atributos y métodos relativos a cada bombilla
    private Estado interruptor;
```

> La parte común a todos los objetos de una clase se representa con miembros de clase y se manejan con métodos de clase.

```
    public Bombilla() {
        interruptor = Estado.OFF;
    }

    public boolean luce() {
        return (interruptor == Estado.ON) &&
               (interruptorGeneral == Estado.ON);
    }

    public void pulsaInterruptor() {
        if (interruptor == Estado.ON){
            interruptor = Estado.OFF;
        }else{
            interruptor = Estado.ON;
        }
    }

    // El main no forma parte del ejercicio
    public static void main(String[] args) {
        System.out.println("Este main no forma parte del ejercicio");

        Bombilla a1 = new Bombilla();
        Bombilla a2 = new Bombilla();
        Bombilla a3 = new Bombilla();

        a1.pulsaInterruptor();
        System.out.println("a1 " + (a1.luce() ? "SI" : "NO") + " luce.");

        a1.pulsaInterruptorGeneral();
        System.out.println("a1 " + (a1.luce() ? "SI" : "NO") + " luce.");

    }
}
```

Comentario: Los métodos que pulsan el interruptor (tanto el general como el de la bombilla) comprueban el estado para cambiarlo, mediante un `if` (o un `switch`).

Si se modela el estado del interruptor con un booleano, llamado `estáPulsado`, el método pulsar se reduce a:

```
public void pulsaInterruptor() {
    estáPulsado = ! estáPulsado;
}
```

Aviso: No sería correcto en este problema representar los interruptores como `boolean`. Es cierto que cada interruptor sólo dispone de dos estados, así como el tipo `boolean` tiene sólo dos valores. Pero las operaciones sobre el interruptor no son las que se dispone de los `boolean`.

Ejercicio 3.8:

Escriba un programa que utilice la clase Bombilla del apartado anterior. Para ello, se creará una Bombilla y se imprimirá por pantalla si luce o no. Luego se pulsa el interruptor de la Bombilla y se vuelve a imprimir el estado de la misma. Por último, se pulsa el interruptor general y se imprime el estado.

Planteamiento: Se define una clase pruebaBombilla. Primero instancia una Bombilla, llamada a1. Como los interruptores pueden tener el valor ON y OFF, con el operador ternario "?:" se imprimen las palabras "SI" o "NO" para determinar si lucen.

Solución:
```
class PruebaBombilla {

    public static void main(String[] args) {
        Bombilla a1 = new Bombilla();

        System.out.println("a1 " + (a1.luce() ? "SI" : "NO") + " luce.");

        a1.pulsaInterruptor();
        System.out.println("a1 " + (a1.luce() ? "SI" : "NO") + " luce.");

        a1.pulsaInterruptorGeneral();
        System.out.println("a1 " + (a1.luce() ? "SI" : "NO") + " luce.");

    }
}
```

Comentario: La salida por pantalla será:

```
a1 NO luce.
a1 NO luce.
a1 SI luce.
```

PAQUETES

Ejercicio 3.9:

Se desea realizar una biblioteca de clases para el tratamiento de imágenes, vídeo y audio para la realización de aplicaciones multimedia. Como existirán multitud de formatos de imágenes, vídeo y sonido, se estructuran las distintas clases en un paquete, llamado mediaBib. Se agruparán los posibles codificadores y decodificadores en tres subpaquetes:

- *"imagen", con los formatos GIF, TIFF y JPEG.*
- *"video" con los formatos H.263, MPEG-1 y MPEG-4*
- *"audio" con los formatos GSM y MP3.*

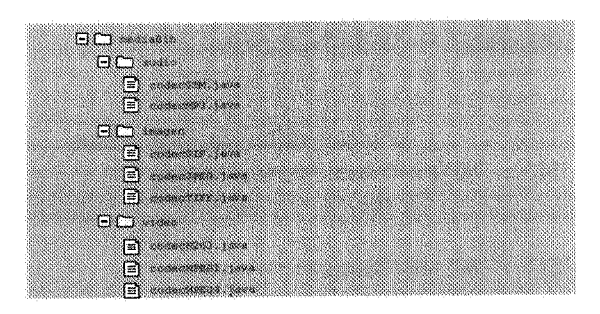
Y, para cada uno de los formatos, se proporcionará una clase que pueda codificar y decodificar dichos formatos. El nombre de esta clase será "codecFormato", sustituyendo "Formato" por el tipo tratado, por ejemplo, codecGIF, codecGSM, etc.

Se pide diseñar la estructura de directorios necesaria para la biblioteca, indicando los directorios y ficheros necesarios. Asimismo, para cada fichero se debe escribir la declaración pertinente respecto de la pertenencia al paquete. NO SE PIDE que se codifiquen las clases, sólo la información de estructura.

Planteamiento: En Java, la estructura de paquetes se ha hecho coincidir con la estructura de carpetas y ficheros en que se organizan las clases, así como las clases públicas han de residir en ficheros con el mismo nombre. Por ello, se generará una carpeta de nombre mediaBib, dentro de la cual habrá tres carpetas más, imagen, video y audio. Por último, dentro de la carpeta imagen se crearán los ficheros codecGIF.java, codecTIFF.java y codecJPEG.java. Dentro de la carpeta vídeo existirán los ficheros codecH263.java,

codecMPEG1.java y codecMPEG4.java. Por último, en la carpeta audio se pondrán los ficheros codecGSM.java y codecMP3.java.

Solución:
La estructura de directorios se muestra a continuación:



Cada fichero debe contener al principio una línea indicando el paquete al que pertenece y luego las clases que aporte. La siguiente lista recoge sólo las líneas de declaración de paquete:

- codecGSM.java: **package** mediaBib.audio;
- codecMP3.java: **package** mediaBib.audio;
- codecGIF.java: **package** mediaBib.imagen;
- codecJPEG.java: **package** mediaBib.imagen;
- codecTIFF.java: **package** mediaBib.imagen;
- codecH263.java: **package** mediaBib.video;
- codecMPEG1.java: **package** mediaBib.video;
- codecMPEG4.java: **package** mediaBib.video;

Comentario: Evidentemente, cada fichero tendrá, al menos, una clase pública con el mismo nombre que el fichero. Los detalles de codificación exceden del propósito de este libro, pero Java Media Framework (JMF) es una biblioteca que cubre estos aspectos.

Aviso: Tenga en cuenta que para la clase que trata el formato H.263 se ha elegido el nombre codecH263. No se puede usar codecH.263, porque los identificadores de clase no pueden llevar un punto. Esto es así porque el punto es el separador para paquetes y generaría ambigüedad un nombre como mediaBib.video.codecH.263. Tampoco se admiten caracteres como el guión '-'. Por razones parecidas, se suelen evitar las peculiaridades de internacionalización, y se suelen evitar acentos y eñes, que sí están admitidos.

CLASES PREDEFINIDAS

Ejercicio 3.10:

Escriba un programa que inicialice dos cadenas de caracteres, una llamada nombre y otra llamada apellido. Los valores iniciales de las cadenas son "Juan" y "Pérez", respectivamente. El programa debe escribir ambas cadenas y su longitud por pantalla.

Planteamiento: Se declaran dos referencias a String, y se inicializan directamente, asignando la cadena de caracteres sin necesidad de new, en la misma línea. Para conocer la longitud de una cadena de caracteres, se usa el método length().

Solución:
```
class Cadenas1 {
```

```
    public static void main (String args[]) {
        String nombre   = "Juan";
        String apellido = "Pérez"; ◄──── El objeto original no se modifica.

        System.out.println(nombre + " tiene " +
                           nombre.length () + " letras");
        System.out.println(apellido + " tiene " +
                           apellido.length () + " letras");
    }
}
```

Comentario: La clase `String` es una de las más usadas de Java. La salida por pantalla del programa es:

```
Juan tiene 4 letras
Pérez tiene 5 letras
```

Alternativamente, una cadena se puede inicializar como `String nombre = new String("Juan")`, expresión completamente equivalente a la anterior, razón por la cual no se usa mucho.

Ejercicio 3.11:

Escriba un programa que declare una cadena de caracteres denominada nombre, con valor inicial "Juan", primerApellido, con valor "Pérez" y segundoApellido con valor "López". Después, el programa concatena a nombre el primer apellido y luego el segundo. Por último, el programa imprime el nombre completo y su longitud.

Planteamiento: Para concatenar dos cadenas, se usa la operación "+". También existe la variante "+=". Para hacer el programa, se crean las tres variables `nombre`, `primerApellido` y `segundoApellido`, inicializadas como indica el enunciado. Luego se concatena con nombre con `primerApellido` con el operador "+=", y luego con `segundoApellido`. Por último, se imprime el nombre y el número de letras.

Solución:

```
class Cadena2 {

    public static void main(String args[]) {
        String nombre = "Juan";
        String primerApellido  = "Pérez";
        String segundoApellido = "López";

        nombre += primerApellido;
        nombre += segundoApellido;

        System.out.println(nombre + " tiene " + nombre.length () + " letras");
    }
}
```

Comentario: Se podría resolver en una sola sentencia la concatenación de los dos apellidos, de la forma:

```
nombre += primerApellido + segundoApellido;
```

La salida del programa es:

```
JuanPérezLópez tiene 14 letras
```

Aviso: Cuando se concatenan cadenas, se forma una nueva, que contiene las dos originales seguidas sin separación ninguna. Si se hubiesen querido blancos entre las subcadenas, hay que ponerlos explícitamente, de la forma:

```
nombre += " " + primerApellido;
nombre += " " + segundoApelldo;
```

Ejercicio 3.12:

Haga un programa que contenga la cadena "Juan Piñón López". El programa ha de imprimir por pantalla ambas cadenas con todos los caracteres en minúsculas y luego en mayúsculas.

Planteamiento: La clase `String` proporciona dos métodos, `toLowerCase()` y `toUpperCase()` que devuelven una cadena con todos los caracteres en minúsculas y en mayúsculas.

Solución:

```
class Cadenas3 {

    public static void main (String args[]) {
        String s1= "Juan Piñón López";

        System.out.println("En minúsculas " + s1.toLowerCase ());
        System.out.println("En mayúsculas " + s1.toUpperCase ());
    }
}
```

Comentario: La salida de este programa es:

```
En minúsculas juan piñón lópez
En mayúsculas JUAN PIÑÓN LÓPEZ
```

Fíjese en que las letras acentuadas y la eñe también han pasado a mayúsculas y minúsculas.

Aviso: Los métodos de la clase `String` no modifican la cadena contenida, sino que devuelven una nueva con las modificaciones deseadas.

Ejercicio 3.13:

Escriba un programa que determine si los dos primeros argumentos pasados en línea son iguales o no.

Planteamiento: Los dos primeros argumentos son `args[0]` y `arg[1]`. Para comparar cadenas, se usa el método `equals()` que devuelve un booleano indicando si son iguales.

Solución:

```
class Cadena4 {

    public static void main (String args[]) {
        if ( args[0].equals(args[1])){
            System.out.println("Son iguales.");
        }else{
            System.out.println("NO son iguales.");
        }
    }
}
```

Comentario: Se podría implementar alternativamente, usando la función compareTo(), que devuelve un 0 si ambas cadenas son iguales. La sentencia quedaría:

```
if ( args[0].compareTo(args[1]) == 0 ){
   System.out.println("Son iguales.");
}else{
   System.out.println("NO son iguales.");
}
```

Aviso: Un error muy común es usar el operador "==" para comparar objetos. Los objetos son referencias, es decir, al comparar con "==" se está evaluando si las referencias apuntan al mismo sitio. El caso de String es más delicado porque, por razones de eficiencia, las cadenas con el mismo valor apuntan al mismo objeto. El siguiente programa:

```
public static void main(String args[]) {
    String s1 = "hola";
    String s2 = "hola";

    if (s1 == s2) {
        System.out.println("Son iguales.");
    } else {
        System.out.println("NO son iguales.");
    }
}
```

imprime por pantalla el mensaje "Son iguales.", con lo que parece que "==" funciona bien. Se aconseja usar siempre el método equals().

Aviso: En este programa habría que comprobar que el array de argumentos tiene al menos dos.

Ejercicio 3.14:

Escriba un método que dado un objeto de la clase String cuente diferentes tipos de caracteres. En particular, el método imprimirá el número de letras, dígitos y espacios en blanco de la cadena. Haga un programa que escriba el conteo de la cadena "Hola, vivo en Marquina 123, 5-7").

Planteamiento: El envoltorio Character tiene funciones que trabajan sobre caracteres. Se usarán los métodos isLetter(), isDigit() e isWhitespace() para comprobar el tipo de un carácter. Para examinar cada carácter del String, se recorre con un bucle for. Pero el bucle for espera una secuencia de un tipo base. Con el método toCharArray() de la clase String, se puede obtener un array de caracteres a partir de un String.

Parámetros: Un String con la cadena a procesar.

Valor de retorno: El método escribe en pantalla, por lo que el valor de retorno se especifica como void.

Solución:

```
class ContarLetras {
    static void conteo (String s) {
       int numLetras   = 0;
       int numDígitos  = 0;
       int numEspacios = 0;
```

```
        for (char c: s.toCharArray()) {
            if  (Character.isLetter(c)){
                numLetras++;
            }else if (Character.isDigit(c)){
                numDígitos++;
            }else if (Character.isWhitespace(c)){
                numEspacios++;
            }
        }

        System.out.println ("Número de letras " + numLetras);
        System.out.println ("Número de dígitos " + numDígitos);
        System.out.println ("Número de espacios " + numEspacios);
    }

    public static void main (String args[]) {
        conteo ("Hola, vivo en Marquina 123, 5-7");
    }
}
```

Se obtiene un array de caracteres para que el for pueda recorrerlo.

Cada tipo de carácter se contabiliza en el apartado correspondiente.

Comentario: La salida del programa quedaría como sigue:

```
Número de letras 18
Número de dígitos 5
Número de espacios 5
```

Otra forma de hacer el recorrido por la cadena sería utilizando el método `charAt(int index)` de la clase `String`, que devuelve el carácter en la posición `index`. El bucle se escribiría así:

```
for (int i = 0; i < s.length(); i++) {
    char c = s.charAt();

    if  (Character.isLetter(c)) {
        numLetras++;
    }else if (Character.isDigit(c)) {
        numDígitos++;
    }else if (Character.isWhitespace(c)) {
        numEspacios++;
    }
}
```

Recorrido con charAt en el String.

Esto no suele usarse por razones de eficiencia y de legibilidad.

Ejercicio 3.15:

Escriba un método que, dado un String, devuelve otro objeto String en el que se cambian todas las vocales minúsculas del original por la letra 'a'.

Planteamiento: El método `replace()` de la clase `String` sustituye un carácter por otro en una cadena. Como se pide cambiar varias letras, se invoca `replace()` sucesivamente, cada vez con una vocal distinta a sustituir por la letra 'a'.

Parámetros: El objeto `String` a modificar.

Valor de retorno: Un valor `String` con las vocales sustituidas.

Solución:

```
String sustituye (String s) {
    s = s.replace('e', 'a');
    s = s.replace('i', 'a');
    s = s.replace('o', 'a');
    s = s.replace('u', 'a');

    return s;

}
```

El objeto original no se modifica.

Aviso: Sólo hay que tener cuidado con no sustituir la vocal 'a', pues es innecesario. El método no sustituye letras acentuadas, etc.

Ejercicio 3.16:

Escriba un método que, dada una cadena de caracteres, devuelve la mitad inicial de la cadena. Escriba un programa que pruebe el método con las cadenas "Hola que tal" y "Adiós".

Planteamiento: El método `substring(inicio, final)` toma dos números y devuelve una cadena que comienza en `inicio` y termina en `final`. Las cadenas comienzan en 0, y `length()/2` da el punto medio. Luego se escribe un `main()` para probar con las cadenas propuestas por el enunciado.

Parámetros: La cadena a dividir, de tipo `String`.

Valor de retorno: Un objeto de la clase `String`, con el resultado.

Solución:

```
class ParteAlMedio {

    static String parteAlMedio(String s) {
        return s.substring(0, s.length() / 2);
    }

    public static void main(String args[]) {
        String s;

        s = "hola que tal";
        System.out.println(s + " " + parteAlMedio(s));

        s = "Adiós";
        System.out.println(s + " " + parteAlMedio(s));
    }

}
```

Se empieza a contar por 0.

Comentario: Este programa escribe por pantalla:

```
hola q
Ad
```

Ejercicio 3.17:

*Escriba un método que, dada una cadena de caracteres, sustituya todas las ocurrencias del texto "es" por "no por". Escriba un segundo método que sustituya todos los grupos de dígitos por un único carácter asterisco, es decir, si la cadena de caracteres es "esto1234es5678bueno900" el primer método debe devolver "no porto1234no por5678bueno900" y el segundo debe devolver "esto*es*bueno*". Escriba el programa que permita comprobar que funciona correctamente.*

Planteamiento: El método `replace(String, String)` toma una expresión regular y la cadena por la que desea sustituir cualquier ocurrencia de la expresión regular. Si el primer parámetro se utiliza una cadena de caracteres se tomará como tal cuando coincida completamente.

Parámetros: La cadena sobre la que se quiere aplicar los cambios, por tanto el parámetro será de la clase `String`.

Valor de retorno: El objeto de la clase `String` modificado.

Solución:

```
public class UsoString {

    public static String cambiaEs(String texto){
        return texto.replaceAll("es", "no por");
    }

    public static String cambiaDigitos(String texto){
        return texto.replaceAll("\\d+", "*");
    }

    public static void main(String[] args) {
        String textoOriginal = "esto1234es5678bueno900";

        System.out.println(cambiaEs(textoOriginal));
        System.out.println(cambiaDigitos(textoOriginal));
    }
}
```

Utiliza una expresión regular que significa un dígito o más.

Comentario: Para ver cómo definir una expresión regular consulte la documentación de la clase `Pattern`. Fíjese que para escribir la expresión regular `\d+` hay que duplicar la barra hacia atrás. El resultado de este programa es:

```
no porto1234no por5678bueno900
esto*es*bueno*
```

Ejercicio 3.18:

Escriba un método que, dada una cadena de caracteres, cuente cuántas veces aparece la misma en dicho texto. Por ejemplo, para la cadena "En un lugar de la mancha, de cuyo nombre no puedo acordarme", si se cuenta cuántas veces aparece "o ", o dicho de otra forma, cuántas palabras acaban en o, debería indicar que son 3.

Planteamiento: Para buscar un texto dentro de una cadena de caracteres se puede utilizar el método `indexOf()` que devuelve la posición en que empieza el texto buscado dentro del original. Para buscarlo más de una vez se usa el mismo método donde el segundo argumento permite indicar a partir de qué posición empezar a buscar.

Parámetros: La cadena en la que se quiere contar cuántas veces aparece el texto y el texto que se quiere buscar. Por tanto, se necesitan dos objetos de la clase `String`.

Valor de retorno: El número de veces que aparece el segundo argumento en el primero, es decir, un valor del tipo `int`.

Solución:
```
public class UsoString {

    public static int cuentaVeces(String texto, String busca){
        int pos = 0;
        int veces = 0;
        while((pos = texto.indexOf(busca, pos)) >= 0){
            pos += busca.length();
            veces++;
        }

        return veces;
    }

    public static void main(String[] args) {

        System.out.println("Veces que aparece \"o \": " +
          cuentaVeces("En un lugar de la mancha, de cuyo nombre no puedo acordarme",
                      "o "));
    }
}
```

Ejercicio 3.19:

Escriba un método que cuente el número de palabras que contiene un texto.

Planteamiento: Para contar el número de palabras se puede utilizar de la clase `String` el método `split()` que permite dividir una cadena de caracteres en fragmentos de acuerdo con una expresión regular, devolviendo un array de `String` con todos los fragmentos del texto original. Para dividir por palabras, se utiliza como expresión regular " +", un espacio y un signo más, que significa un espacio una o más veces. Antes de llamar a `split()` se llamará al método `trim()` para eliminar los espacios en blanco que pueda haber al principio y al final del texto.

Parámetros: La cadena de la que se quiere contar el número de palabras que tiene.

Valor de retorno: El número de palabras de la cadena de caracteres, es decir, un valor del tipo `int`.

Solución:
```
public class UsoString {

    public static int cuentaPalabras(String texto){
        String[] palabras = texto.trim().split(" +");
        return palabras.length;
    }

    public static void main(String[] args) {
        System.out.println("Número de palabras: " +
```

Utiliza una expresión regular que significa un espacio o más.

```
            cuentaPalabras("En un lugar de la mancha, de cuyo nombre no puedo acordarme"));
    }
}
```

Ejercicio 3.20:

Escriba un programa que determine cuántas veces una persona celebra su cumpleaños en lunes. El programa debe imprimir las fechas en que el cumpleaños cae en lunes los primeros 40 años de su vida y el total de ellas. Suponga el ejemplo de una persona que nació el 4 de abril de 1965 y tiene 40 años.

Planteamiento: Mediante la clase `GregorianCalendar`, se construye la fecha deseada. Con un bucle `for` se recorren los 40 años, añadiendo cada año mediante el método `roll()`. Para cada fecha, se obtiene el día de la semana y se compara si es lunes (`Calendar.MONDAY`), en cuyo caso se imprime la fecha en formado día/mes/años. Además, se proporcionará un contador

Solución:

```
import java.util.GregorianCalendar;
import java.util.Calendar;

public class CuentaDías {

    public static void main(String[] args) {
        int cuentaLunes = 0;

        GregorianCalendar gc = new GregorianCalendar(1965, Calendar.APRIL, 4);

        for (int i = 0; i < 40; i++) {
            gc.roll(Calendar.YEAR, 1);          <-- Avanza un año.

            if (gc.get(Calendar.DAY_OF_WEEK) == Calendar.MONDAY) {
             cuentaLunes++;

             System.out.println("El " + gc.get (Calendar.DAY_OF_MONTH) +
                                "/"  + (gc.get (Calendar.MONTH)+1) +     <-- Los meses se representan desde 0
                                "/"  + gc.get (Calendar.YEAR) +              (enero) a 11 (diciembre).
                                " es lunes");
            }
        }

        System.out.println ("El cumpleaños se celebra  en " + cuentaLunes + " lunes");
    }
}
```

Comentario: La salida del programa es:

```
El 4/4/1966 es lunes
El 4/4/1977 es lunes
El 4/4/1983 es lunes
El 4/4/1988 es lunes
El 4/4/1994 es lunes
El 4/4/2005 es lunes
El cumpleaños se celebra en 6 lunes
```

Aviso: El bucle `for` primero incrementa el año (método `roll`) porque se considera que el día de nacimiento no se celebra el cumpleaños.

Hay que tener cuidado con los meses, pues `Calendar` considera el 0 como enero y el 11 como diciembre. Si se escribe `GregorianCalendar(1965, 4, 4)`, se especifica el 4 de mayo de 1965.

IMPORTACIÓN ESTÁTICA

Ejercicio 3.21:

Escriba un método que dado un ángulo en radianes, lo escriba en pantalla en grados, así como los valores del seno, el coseno y la tangente. Las funciones de Math han de importarse de forma estática. Haga un programa que imprima los valores de esas tres funciones para los valores 0, $\pi/4$, $\pi/2$, y π en radianes.

Planteamiento: Al importar de forma estática, no se requiere anteponer el nombre de la clase. Entonces se importa toda la clase `Math`. El ángulo, cuando no se dice nada, se supone que está en radianes. El método `toDegrees()` lo pasa a grados.

Parámetros: Un `double` con el ángulo.

Valor de retorno: El resultado del método es escribir en pantalla. El método no necesita devolver ningún valor de retorno por lo que se utiliza la cláusula `void`.

Solución:

```
import static java.lang.Math.*;

public class Trigonometría {

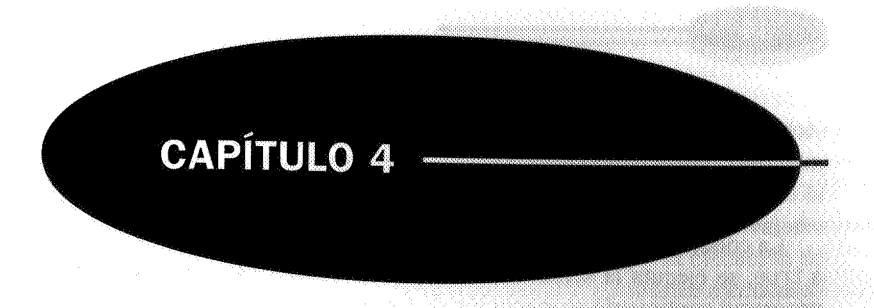
    static void funciones(double alfa) {
        System.out.println(toDegrees(alfa) + " " + sin(alfa) +
                           " " + cos(alfa) + " " + tan(alfa));
    }
    public static void main(String args[]) {
        funciones(0);
        funciones(PI / 4);
        funciones(PI / 2);
        funciones(PI);
    }
}
```

Importación estática de todos los métodos de la clase `Math`.

Comentario: La salida del programa quedaría como sigue:

```
0.0 0.0 1.0 0.0
45.0 0.7071067811865475 0.7071067811865476 0.9999999999999999
90.0 1.0 6.123233995736766E-17 1.633123935319537E16
180.0 1.2246467991473532E-16 -1.0 -1.2246467991473532E-16
```

Aviso: El código queda más simple, pero no siempre mejora la legibilidad del programa, porque se olvida de dónde proceden los métodos.



Estructuras de control

4.1 ESTRUCTURAS DE SELECCIÓN

4.1.1 Estructura if

La estructura if se escribe de la siguiente forma:

```
if (condición) {
   sentencias
}
```

La condición se escribe siempre entre paréntesis. La condición es una expresión que evalúa un valor de tipo boolean (booleano) como las expresiones aritmético-lógicas vistas en el Capítulo 1. Si la condición devuelve true se ejecuta el conjunto de sentencias dentro del bloque del if.

Un bloque de sentencias está delimitado por llaves, por lo que la estructura if está asociada a todo el bloque de sentencias. Si la estructura if lleva asociada sólo una sentencia, se puede poner sin llaves, pero se recomienda añadirlas aunque sea redundante. Esto mismo es aplicable al resto de estructuras que se describen en las siguientes secciones.

Por ejemplo, suponga que en un programa existe una variable velocidad de tipo int cuyo valor es 80:

```
if (velocidad <= 90){
    System.out.println("La velocidad es menor o igual que 90.");
}
```

Como la condición vale true, se escribe por pantalla el mensaje de la sentencia dentro del bloque if.

4.1.2 Estructura if-else

La estructura if-else se escribe de la siguiente forma:

```
if (condición) {
   sentencias
```

```
} else {
   sentencias
}
```

La condición es una expresión lógica como en la sección anterior. Si la condición es verdadera, es decir, se evalúa a true, se ejecuta la sentencia o bloque de instrucciones que se encuentra a continuación del if. Seguidamente, el programa continuaría su ejecución de forma secuencial con la sentencia inmediatamente posterior a **toda** esta estructura. Si, por el contrario, la condición se evalúa a false, se ejecuta la sentencia o el bloque de sentencias que se encuentra a continuación del else y el programa continuaría su ejecución.

Por ejemplo, suponga que en un programa existe una variable velocidad de tipo int cuyo valor es 110:

```
if (velocidad <= 90) {
    System.out.println("La velocidad es menor o igual que 90.");
} else {
    System.out.println("Velocidad excesiva. Supera los 90.");
}
```

⇨ Sobre la estructura if-else consulte los Ejercicios 4.1 a 4.11.

4.1.3 Operador condicional

El operador condicional **(?:)** está relacionado con la estructura if-else. El operador evalúa la condición a la izquierda del símbolo ?. Si la condición vale true devuelve el valor de la expresión que haya entre el ? y el :. Si la condición vale false se devuelve el valor de la expresión tras el símbolo :. Por ejemplo, para escribir el mayor de tres números se puede escribir.

```
System.out.print("El número mayor es el de valor ");
System.out.println(a > b ? a : b);
```

⇨ Sobre el operador condicional consulte los Ejercicios 4.2 a 4.6 y 4.16.

4.1.4 Estructura switch

La estructura switch consta de una expresión y una serie de etiquetas case con un caso opcional default. La sentencia switch se escribe de la siguiente forma:

```
switch (expresión){
        case valor1:
                sentencias;
                break;
        case valor2:
        case valor3:
                sentencias;
                break;
        ...
        default:
                sentencias;
                break;
}
```

La expresión, que es obligatorio que esté entre paréntesis, tiene que evaluarse a un entero, un carácter, un enumerado o un boolean. A continuación, en cada `case` aparece un valor que únicamente puede ser una expresión constante, es decir, una expresión cuyo valor se puede conocer antes de empezar a ejecutar el programa, del mismo tipo que la expresión del `switch`. Después de cada `case` se puede poner una única sentencia o un conjunto de ellas. Los valores asociados en cada `case` se comparan en el orden en que están escritos. Cuando se quiere interrumpir la ejecución de sentencias se utiliza la sentencia `break` que hace que el control del programa termine el `switch` y continúe ejecutando la sentencia que se encuentre después de esta estructura. Si no coincide el valor de ningún `case` con el resultado de la expresión, se ejecuta la parte `default`.

Si ningún valor de los `case` coincide con el resultado de la expresión y la parte `default` no existe, ya que es opcional, no se ejecuta nada de la estructura `switch`.

Por ejemplo, suponga que desea conocer el número de días de un mes dado. Un switch es la mejor opción para seleccionar cómo calcular cuántos días le corresponden a cada mes.

```
public class DíasDelMes {
    enum Mes {Enero, Febrero, Marzo, Abril,
              Mayo, Junio, Julio, Agosto,
              Septiembre, Octubre, Noviembre, Diciembre}

    public static void main(String[] args) {
        int días;
        Mes mes = Mes.Noviembre;
        switch (mes) {
            case Abril:
            case Junio:
            case Septiembre:
            case Noviembre:
                días = 30;
                break;
            case Febrero: // no se conoce el año
                días = 28;
                break;
            default:
                días = 31;
                break;
        }
        System.out.println("El mes " + mes + " tiene " + días + " días.");
    }
}
```

⇨ Sobre la estructura switch consulte los Ejercicios 4.13 y 4.24.

4.2 ESTRUCTURAS DE REPETICIÓN

4.2.1 Estructura while

La estructura de repetición `while` sigue el siguiente esquema:

```
while (condición){
   sentencias
}
```

La condición tiene que estar obligatoriamente entre paréntesis. La condición es una expresión lógica. Si la condición vale true, se ejecutan las sentencias que componen el bucle. Cuando concluye la ejecución de las instrucciones del bucle se vuelve a evaluar la condición. De nuevo, si la condición es cierta se vuelven a ejecutar las instrucciones del bucle. En algún momento la condición valdrá false, en cuyo caso finaliza la ejecución del bucle y el programa continúa ejecutándose por la sentencia que se encuentre a continuación de la estructura while.

Por ejemplo, el siguiente método devuelve true si el número introducido por parámetro es un número primo.

```
public boolean primo(int numero){
   int divisor=2;
   boolean primo = true;

   while ((divisor * divisor <= numero) && primo) {
      if (numero % divisor == 0)
         primo = false;
      divisor++;
   }

   return  primo;
}
```

⇨ Sobre la estructura while consulte los Ejercicios 4.13, 4.14 y 4.24.

4.2.2 Estructura do-while

La sentencia do-while tiene la siguiente estructura:

```
do {
   sentencias
} while (condición);
```

Cuando en un programa se llega a una estructura do-while se empiezan a ejecutar las sentencias que componen la estructura. Cuando se terminan de ejecutar se evalúa la condición. Si la condición vale true, se ejecutan de nuevo las sentencias que componen la estructura. El bucle deja de repetirse cuando, tras evaluarse la condición, ésta vale false. Como ocurría con la estructura while es necesario que dentro del bucle, en algún momento, la condición valga false para evitar que el bucle se ejecute indefinidamente.

Suponga que se desea realizar un programa que solicite al usuario un número entre 0 y 100, ambos inclusive. Al terminar debe imprimir por pantalla el número introducido. Si el número introducido no está dentro del intervalo, debe dar un mensaje de error. Un posible programa sería el que se muestra en el siguiente ejemplo:

```
import java.util.Scanner;

public class PedirNumero {
    public static void main(String[] args) {
        int numero;
        String linea;

        Scanner teclado = new Scanner(System.in);

        do {
            System.out.print("Introduzca un número entre 0 y 100: ");
```

```
            numero = teclado.nextInt();
        } while (numero < 0 || numero > 100);

        System.out.println("El número introducido es: " + numero);
    }
}
```

⇨ Sobre la estructura do-while consulte los Ejercicios 4.28, 4.30 y 4.51.

4.2.3 Estructura for

La estructura `for` tiene dos formas. La más habitual es:

```
for (inicialización; condición; actualización){
   sentencias
}
```

Los elementos de que consta son los siguientes:

- La `inicialización` es una sentencia que permite inicializar el bucle, puede ser la declaración e inicialización de las variables que se utilizan en el bucle. Esta sentencia de inicialización se ejecuta únicamente una vez en la primera ejecución del bucle.
- La `condición` es la condición para continuar la ejecución del bucle. La condición se evalúa siempre antes de empezar a ejecutar el bucle. Si la condición es cierta, se ejecutan las sentencias del bucle.
- Después de ejecutar las sentencias del bucle y antes de volver a evaluar la condición se ejecuta la `actualización`. Esta parte se suele utilizar para modificar el valor de las variables que forman parte de la condición.

Todos los elementos que se acaban de describir del bucle `for` son opcionales, es decir, pueden ser vacíos. Por ejemplo, para escribir por pantalla 5 veces el mensaje "Hola a todos" con el número de vez, se puede escribir:

```
for (int i = 0; i < 5; i++) {
   System.out.print(i + 1);
   System.out.println(" Hola a todos.");
}
```

La segunda estructura de un bucle `for` es la siguiente:

```
for (variable : estructura){
   sentencias
}
```

donde la variable indicada entre paréntesis, que se puede declarar en ese mismo lugar, va tomando el valor de todas las variables de la estructura indicada, repitiéndose el bucle para todos los valores. Por ejemplo, para escribir por pantalla todos los elementos de un array de String se puede escribir:

```
for(String a : miArrayDeString){
    System.out.println(a);
}
```

⇨ Sobre la estructura for consulte los Ejercicios 4.15 a 4.27.

4.2.4 Uso de las estructuras de repetición

Es importante utilizar el tipo de bucle más apropiado en cada parte de un programa. En la Tabla 4.1 se pueden ver las reglas a tener en cuenta para usar una u otra estructura de repetición.

Tabla 4.1. Reglas de uso de las estructuras de repetición.

Estructura	Usar si
`while`	Si el bucle se ejecuta 0 o más veces. Es decir, si hay casos donde no se ejecute.
`do-while`	Si la parte de ejecución del bucle se ha de hacer al menos una vez.
`for`	Se sabe el número de veces que se ha de repetir el bucle. Si utilizar la inicialización y la actualización del bucle permite escribir el código de forma más clara. Se realiza un recorrido en una estructura de almacenamiento. Si la estructura de almacenamiento se va a recorrer completa realizando operaciones con sus valores, se utilizará la segunda versión del `for`.

4.3 ESTRUCTURAS DE SALTO

4.3.1 Sentencia break

La sentencia `break` se utiliza para terminar inmediatamente la ejecución de una estructura de repetición o de un `switch`. Una vez se ha ejecutado la sentencia `break`, la ejecución continúa tras la estructura de repetición o `switch` donde se ha ejecutado `break`.

⇨ Sobre la sentencia break consulte los ejercicios de uso de la sentencia switch y el 4.24.

4.3.2 Sentencia continue

La sentencia `continue` únicamente puede aparecer dentro de una estructura de repetición. El efecto que produce es que se deja de ejecutar el resto del bucle para volver a evaluar la condición del bucle, continuando con la siguiente iteración si el bucle lo permite.

4.4 EXCEPCIONES

4.4.1 Captura

La estructura de captura de excepciones es `try-catch-finally`.

```
try {
    sentencias
} catch (ClaseException e) {
    sentencias
} catch (ClaseException e) {
```

```
    sentencias
} catch (ClaseException e) {
    sentencias
} finally {
    sentencias
}
```

En la estructura try-catch-finally, la parte catch puede repetirse tantas veces como excepciones diferentes se deseen capturar. La parte finally es opcional y sólo puede aparecer una vez.

Para ver cómo funciona esta estructura en el siguiente ejemplo se presenta un programa que pide al usuario un número entre 0 y 100, con manejo de excepciones.

```
import java.io.*;

public class PedirNumero2 {
    public static void main(String[] args) {
        int numero = -1;
        int intentos = 0;
        String linea;

        BufferedReader teclado = new BufferedReader(new InputStreamReader(System.in));

        do {
            try{
                System.out.print("Introduzca un número entre 0 y 100: ");
                linea  = teclado.readLine();
                numero = Integer.parseInt(linea);
            }catch(IOException e){
                System.out.println("Error al leer del teclado.");
            }catch(NumberFormatException e){
                System.out.println("Debe introducir un número entero entre 0 y 100.");
            }finally{
                intentos++;
            }
        } while (numero < 0 || numero > 100);

        System.out.println("El número introducido es: " + numero);
        System.out.println("Numero de intentos: " + intentos);
    }

}
```

En este programa se pide repetidamente un número utilizando una estructura do-while, mientras el número sea menor que 0 o sea mayor que 100. Dentro de esta estructura do-while es donde se solicita el número al usuario. El método readLine() de la clase BufferedReader puede lanzar la excepción IOException. Si no se hace nada con ella, cuando se produce el programa termina. Así mismo, la llamada al método parseInt() puede lanzar la excepción NumberFormatException. Esta excepción se lanza cuando parseInt() no puede convertir el texto en un número entero.

Cuando se produce una excepción se compara si coincide con la excepción del primer catch. Si no coincide se compara con la excepción del segundo catch, y así sucesivamente. Cuando se encuentra un catch cuya excepción coincide con la que se ha lanzado, se ejecuta el bloque de sentencias de ese catch.

Si ningún bloque catch coincide con la excepción lanzada, dicha excepción se lanza fuera de la estructura try-catch-finally. Si no se captura en el método, se debe lanzar delegándola. Para ello debe declararlo en la cabecera del método.

El bloque finally se ejecuta tanto si try terminó normalmente como si se capturó una excepción en algún bloque catch. Es decir, el bloque finally se ejecuta siempre.

4.4.2 Delegación

Se produce una delegación de excepciones cuando un método utiliza una sentencia que puede generar una excepción, pero la excepción que se puede generar no la captura y la trata, sino que la delega a quien le llamó. Para ello, simplemente, declara en la cabecera que la excepción que se puede producir durante la ejecución del método la puede generar dicho método de la siguiente forma:

```
public class Alumno {
    ...
    public int leeAño(BufferedReader lector) throws IOException, NumberFormatException {
        String linea = teclado.readLine();
        Return Integer.parseInt(linea);
    }
    ...
}
```

De esta forma si al leer del BufferedReader que recibe en el parámetro se encuentra con algo que no es un entero, se generará una excepción y en lugar de devolver el entero, se enviará la excepción a donde se llamó a este método.

4.4.3 Definición de excepciones de usuario

Una excepción es una clase. Para crear una excepción se deriva de la clase Exception. Para más detalles sobre derivación de clases y herencia véase el Capítulo 5. Si se desea tener una excepción que indique un año fuera de rango, se puede hacer de la siguiente forma:

```
public class AñoFueraDeRangoException extends Exception {
    public AñoFueraDeRangoException(String texto) {
        super(texto);
    }
}
```

Al crear la excepción, se ha añadido un constructor que acepta un String. De esta forma se puede añadir un mensaje al crear la excepción que dé detalles adicionales sobre el problema que ha generado la excepción. A partir de este momento se dispone de una excepción que se puede lanzar cuando sea necesario.

4.4.4 Lanzamiento de excepciones de usuario y redefinición

Si se dispone de la excepción declarada en el apartado anterior en una posible clase Alumno, el método ponAñoDeNacimiento() quedaría:

```
public class Alumno {
    ...
```

```
        public void ponAñoDeNacimiento(int año) throws AñoFueraDeRangoException {
            if (año < 1900 || año > 1990)
                throw new AñoFueraDeRangoException("Demasiado joven o demasiado viejo.");
            añoDeNacimiento = año;
        }
        ...
    }
```

De esta forma, si al llamar al método se utiliza un valor de año anterior a 1900 o posterior a 1990, el método no asigna ese año al alumno, sino que lanza una excepción indicando que ese año no tiene sentido para un alumno. Para lanzar una excepción se utiliza la sentencia `throw` con el objeto excepción que se desea lanzar. Se ha hecho en el mismo paso la creación y lanzamiento de la excepción.

Para capturar dicha excepción desde donde se llamó al método `ponAñoDeNacimiento()`, se haría entonces de la siguiente forma:

```
    try {
        ...
        alumno.ponAñoDeNacimiento(unAño);
        ...
    } catch (AñoFueraDeRangoException e) {
        // manejar la excepción de poner el año
        ...
    }
```

⇨ Sobre el uso de excepciones en sus distintas formas consulte los Ejercicios del 4.30 al 4.51.

4.5 ASERCIONES

4.5.1 Aserciones como comprobación de invariantes

Una aserción tiene la siguiente sintaxis:

```
    assert condicion;
```

o

```
    assert condicion : mensaje;
```

donde la condición es una expresión lógica que debe cumplirse en el momento en que se evalúa la aserción. Si la condición no se cumple, el programa termina con un error. Tras el operador dos puntos puede ponerse un String con el mensaje que se devuelve como error.

4.5.2 Aserciones como precondiciones

Como convenio, las precondiciones de un método público en Java es preferible que se comprueben mediante una condición y lancen la excepción `IllegalArgumentException` o la excepción apropiada de acuerdo con el error encontrado, como en el siguiente ejemplo:

```
    public void ponEdad(int edad) throws Exception{
```

```
        if(edad < 0 || edad > 150)
            throw new Exception("Edad no válida.");

        //resto del método
        ...
    }
```

Sin embargo, en un método no público sí se puede sustituir esa comprobación, en muchos casos redundante, por una comprobación que asegure que el método se utiliza de forma consistente en la clase, como en el siguiente ejemplo.

```
    private void fijaIntervalo(int intervalo){
        assert intervalo > 0 && intervalo <= MAX_INTERVALO;

        //resto del método
        ...
    }
```

Si al llamar al método `fijaIntervalo()`, el valor del parámetro no está dentro del intervalo establecido, al no cumplirse la condición de la aserción, el programa terminará con un error.

4.5.3 Aserciones como postcondiciones

En este caso sí se recomienda el uso de aserciones para la comprobación de las postcondiciones de un método, pues se supone que el método se encuentra bien implementado. En este caso, el uso de postcondiciones permite asegurar que se cumplen en todas las ejecuciones que se hacen del método.

```
    public void ordenar(int[] datos){
        //algoritmo de ordenación
        ...

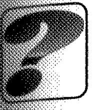
        boolean desordenado = false;
        for(int i=0; i<datos.length-1; i++)
            desordenado = desordenado || datos[i]>datos[i+1];
        assert !desordenado : "Ordenar: array no ordenado.";
    }
```

El método anterior es un método de ordenación de números enteros. Al finalizar el método debe asegurarse que el array termina con todos sus datos ordenados. Como esta comprobación es compleja se ha añadido unas líneas de código que hacen la comprobación. En la aserción se ha utilizado la versión completa, en la que si la condición no se cumple, se evalúa la parte que hay tras los dos puntos y con ello se construye el error `AsertionError`. Cuando el programa termina, aparecerá en pantalla el mensaje que se ha puesto tras los dos puntos.

4.5.4 Aserciones como invariantes

Las aserciones como invariantes permiten comprobar en cualquier parte del código de un método que se cumplen los invariantes establecidos. Su uso es el mismo que para las postcondiciones.

⇨ Sobre el uso de aserciones consulte el Ejercicio 5.10.



Problemas resueltos

Estructuras de selección

Ejercicio 4.1:

Escriba un método, de nombre mayorDeEdad, que reciba una edad como entero por parámetro y muestre un mensaje por pantalla si es mayor de edad o no.

Planteamiento: El método debe escribir "La persona es mayor de edad" o "La persona no es mayor de edad". Se utilizará la estructura `if` para añadir dentro del mensaje de salida "no" si no se cumple que el valor proporcionado al método como parámetro es mayor que el valor constante 18.

Parámetro: El método recibe por parámetro un valor entero, que se comparará con la constante también entera 18.

Valor de retorno: El método no necesita devolver ningún valor de retorno por lo que se utiliza la cláusula `void`.

Solución:

```
public void mayorDeEdad(int edad) {
    final int MAYORIA_EDAD = 18;

    System.out.print("La persona ");
    if(edad < MAYORIA_EDAD){
        System.out.print("no");
    }
    System.out.println(" es mayor de edad.");
}
```

Ejercicio 4.2:

Escriba un método, de nombre mayorQue25, que reciba un entero por parámetro y muestre un mensaje por pantalla que indique si es mayor o menor que 25.

Planteamiento: El método se podría realizar, igual que el anterior, utilizando la estructura `if-else`. Pero también es posible implementar una solución utilizando el operador condicional (? :) .

Parámetro: El método recibe por parámetro un valor entero (`int`), que se comparará con la constante también entera 25.

Valor de retorno: El método no necesita devolver ningún valor de retorno por lo que se utiliza la cláusula `void`.

Solución:

```
public void esMayorQue25(int n) {
    final int VEINTICINCO = 25;

    System.out.print("El número " + n);
    System.out.print(n > VEINTICINCO ? " es ":" no es ");
    System.out.println("mayor que " + VEINTICINCO + ".");
}
```

Ejercicio 4.3:

Escriba un método, de nombre mayorDeEdad, que reciba una edad como entero por parámetro y devuelva si la persona es mayor de edad o no.

Planteamiento: Se proponen dos soluciones. El método, al igual que el anterior, se podría realizar utilizando la estructura `if`. Pero también es posible implementar una solución utilizando el operador condicional (**?:**). En ambos casos el método debe ejecutar una sentencia `return` con el valor booleano adecuado.

Parámetro: El método recibe por parámetro un valor entero, que se comparará con la constante también entera 18.

Valor de retorno: El método devuelve un valor `boolean`.

Solución1: (con sentencia if)

```
public boolean mayorDeEdad(int edad){
    final int MAYORIA_EDAD = 18;

    if (edad >= MAYORIA_EDAD){
        return true;
    }
    return false;
}
```

Solución2: (con operador condicional)

```
public boolean mayorDeEdad(int edad){
    final int DIECIOCHO = 18;

    return (edad >= DIECIOCHO) ? true : false;
}
```

Ejercicio 4.4:

Escriba un programa, de nombre pares, en el que se solicite un número entero al usuario y el programa escribirá un mensaje por pantalla que indique si se trata de un número par o de un número impar.

Planteamiento: Después de leer del teclado el número solicitado al usuario, para comprobar que se trata de un número par o impar, se realiza la operación módulo 2. Si el valor del módulo es 0 se trata de un número par y se escribe el correspondiente mensaje, si no se escribirá el mensaje correspondiente a un número impar. Esto se realiza mediante la estructura `if-else`.

Solución:

```
import java.util.*;

public class ParesImpares{
    public static void main(String[] args){
        Scanner teclado = new Scanner(System.in);

        System.out.print("Escriba un número entero: ");
        int número = teclado.nextInt();

        if(número % 2 == 0){
            System.out.println("El número " +  número + " es par.");
```

```
        }else{
            System.out.println("El número " + número + " es impar.");
        }
    }
}
```

Comentario: Recuerde que un número es par si es divisible por dos e impar en caso contrario.

Ejercicio 4.5:

Escriba un método, de nombre escribeOrdenadosDosNúmeros, que reciba por parámetro dos números reales y los escriba de menor a mayor.

Planteamiento: Se proponen dos formas de implementación, mediante el operador ternario y mediante la estructura `if-else`.

Parámetros: El método recibe como argumentos los dos valores reales (`double`) que se quieren imprimir ordenados.

Valor de retorno: El método no necesita devolver ningún valor de retorno por lo que se utiliza la cláusula `void`.

Solución 1:

```
public void escribeOrdenadosDosNúmeros (double a, double b){
    System.out.println(a < b ?  a + " " + b : b + " " + a );
}
```

Solución 2:

```
public void escribeOrdenadosDosNúmeros(double a, double b){
    if(a < b){
        System.out.println(a + " " + b);
    }else{
        System.out.println(b + " " + a);
    }
}
```

Ejercicio 4.6:

Escriba un método, de nombre escribeOrdenadosTresNúmeros, que reciba por parámetro tres números reales y los escriba de menor a mayor.

Planteamiento: Se pueden comparar los tres valores entre sí para imprimirlos en el orden adecuado. En primer lugar se comprueba cuál es el menor que los otros dos y una vez tomada esa decisión se comprueba entre los dos restantes.

Parámetros: El método recibe como argumentos los tres valores reales (`double`) que se quieren imprimir ordenados.

Valor de retorno: El método no necesita devolver ningún valor de retorno por lo que se utiliza la cláusula `void`.

Solución 1:

```
public void escribeOrdenadosTresNúmeros(double a, double b, double c){
```

```
        System.out.println((a < b && a < c) ?
                              b < c ? a + " " + b + " " + c :
                                      a + " " + c + " " + b :
                            (b < a && b < c) ?
                              a < c ? b + " " + a + " " + c :
                                      b + " " + c + " " + a :
                            (c < a && c < b) ?
                              a < b ? c + " " + a + " " + b :
                                      c + " " + b + " " + a : ""
                              );
}
```

Planteamiento: También se pueden ordenar los valores antes de imprimirlos. Para ordenar los valores se comparan de dos en dos y, si es necesario, se intercambian sus valores mediante una variable auxiliar.

Solución 2:

```
public void escribeOrdenadosTresNúmeros_2(double a, double b, double c){
    double temp;

    if (a > b){
        temp = a;
        a = b;
        b = temp;
    }

    if (a > c){
        temp = a;
        a = c;
        c = temp;
    }

    if (b > c){
        temp = b;
        b = c;
        c = temp;
    }

    System.out.printf("%f %f %f\n", a, b, c);
}
```

Comentario: Aunque como se ve, es factible una implementación con el operador condicional, se recomienda la segunda solución por ser mucho más fácil de leer.

Ejercicio 4.7:

Escriba un programa, de nombre DosPersonas, que pida el nombre y dos apellidos de dos personas y los escriba ordenados alfabéticamente, teniendo en cuenta los dos apellidos y, si fuese necesario, el nombre.

Planteamiento: El programa debe solicitar los datos de las dos personas. Compara el primer apellido de ambas personas y si son distintos se utiliza el orden de impresión por primer apellido. Si el primer apellido coincide se comparan por el segundo apellido. Si también coinciden se comparan por el nombre.

Solución:

```
import java.util.*;

public class Ejercicio11 {
    public static void main(String[] args){
        Scanner teclado = new Scanner(System.in);

        System.out.println("El programa le pedirá el nombre y dos apellidos de dos personas.");
        System.out.print("Introduzca el nombre de la primera persona: ");
        String nombre1 = teclado.nextLine();
        System.out.print("Introduzca el primer apellido: ");
        String apellido1Per1 = teclado.nextLine();
        System.out.print("Introduzca el segundo apellido: ");
        String apellido2Per1 = teclado.nextLine();

        System.out.print("Introduzca el nombre de la segunda persona: ");
        String nombre2 = teclado.nextLine();
        System.out.print("Introduzca el primer apellido: ");
        String apellido1Per2 = teclado.nextLine();
        System.out.print("Introduzca el segundo apellido: ");
        String apellido2Per2 = teclado.nextLine();

        if((apellido1Per1.compareToIgnoreCase (apellido1Per2) < 0) ||
            (apellido1Per1.equalsIgnoreCase(apellido1Per2) &&
                apellido2Per1.compareToIgnoreCase(apellido2Per2) < 0) ||
            (apellido1Per1.equalsIgnoreCase(apellido1Per2) &&
                apellido2Per1.equalsIgnoreCase(apellido2Per2) &&
                nombre1.compareToIgnoreCase(nombre2) < 0)){
            System.out.println(apellido1Per1 + " " + apellido2Per1 + ", "+ nombre1);
            System.out.println(apellido1Per2 + " " + apellido2Per2 + ", "+ nombre2);
        }else{
            System.out.println(apellido1Per2 + " " + apellido2Per2 + ", "+ nombre2);
            System.out.println(apellido1Per1 + " " + apellido2Per1 + ", "+ nombre1);
        }

    }
}
```

compareToIgnoreCase() no tiene en cuenta mayúsculas y minúsculas.

Ejercicio 4.8:

Escriba un método, de nombre cambioEnMonedas, que reciba como parámetro un valor real que indica una cantidad de dinero en euros. El método imprime por pantalla la cantidad de monedas de cada tipo en que se debe devolver la cantidad de dinero indicada.

Planteamiento: Al tratarse de una cantidad de dinero en euros, al principio el valor del argumento debe tener como máximo dos decimales. Para ello, se multiplica el valor por 100 y se opera con el valor como un número entero. Esta operación consigue que siempre se opere en céntimos. El número de monedas que se deben devolver se calcula empezando con la de mayor valor y continuando con las siguientes: 2 euros (200 céntimos), 1 euro (100 céntimos), 50 céntimos, 20 céntimos, 5 céntimos, 2 céntimos y por último 1 céntimo. Para realizar las operaciones se utiliza la división entera y la operación resto.

Parámetros: Recibe como argumento un valor real, con dos decimales como máximo.

Valor de retorno: El resultado de las operaciones se escribe directamente en pantalla. El método no necesita devolver ningún valor de retorno por lo que se utiliza la cláusula `void`.

Solución:

```
public void escribeCambioEnMonedas(double cambio){
    int devolver = (int)(cambio * 100); // en céntimos.

    System.out.println("Para devolver " + cambio + " debe dar:");
    // para cada moneda
    if (devolver >= 200){
        System.out.println("Monedas de 2 euros: " + devolver/200);
        devolver %= 200;
    }
    if (devolver >= 100){
        System.out.println("Monedas de 1 euro: " + devolver/100);
        devolver %= 100;
    }
    if (devolver >= 50){
        System.out.println("Monedas de 50 centimos: " + devolver/50);
        devolver %= 50;
    }
    if (devolver >= 20){
        System.out.println("Monedas de 20 centimos: " + devolver/20);
        devolver %= 20;
    }
    if (devolver >= 10){
        System.out.println("Monedas de 10 centimos: " + devolver/10);
        devolver %= 10;
    }
    if (devolver >= 5){
        System.out.println("Monedas de 5 centimos: " + devolver/5);
        devolver %= 5;
    }
    if (devolver >= 2){
        System.out.println("Monedas de 2 centimos: " + devolver/2);
        devolver %= 2;
    }
    if (devolver >= 1){
        System.out.println("Monedas de 1 centimo: " + devolver);
        devolver = 0;
    }
}
```

Comentario: El método únicamente se ha planteado para que escriba el cambio en monedas, por lo que si el valor que se pasa como argumento es muy elevado mostrará una cantidad muy alta de monedas de 2 euros.

Ejercicio 4.9:

Escriba un programa, de nombre Socio, que calcule la cuota que se debe abonar en el club de golf. La cuota es de 500 euros. Tendrán un 50% de descuento las personas mayores de 65 años y un 25% los menores de 18 años si los padres no son socios y 35% si los padres son socios.

Planteamiento: El programa debe solicitar la edad del usuario si ésta es mayor de 65. Al precio del abono se le aplica el descuento del 50%. Si la edad es menor que 18, se pregunta al usuario si su padre es socio del club. Si la respuesta es sí, se le aplica un descuento del 35%; en caso contrario, se le aplica un descuento del 20%. Una vez calculado el precio del abono se escribe por pantalla.

Solución:

```
import java.util.*;

public class Socio{
    public static void main(String[] args){
        final int CUOTA = 500; // en euros
        final int PORCENTAJE_MAYORES = 50; // por ciento
        final int PORCENTAJE_JOVEN_SOCIO = 35; // por ciento
        final int PORCENTAJE_JOVEN_NO_SOCIO = 25; // por ciento

        int edad, cuota;
        Scanner teclado = new Scanner(System.in);

        System.out.print("Introduzca la edad del nuevo socio: ");
        edad = teclado.nextInt();

        cuota = CUOTA;

        if(edad >= 65){
            cuota -= CUOTA * PORCENTAJE_MAYORES / 100;
        }else if (edad < 18){
            System.out.print("¿El padre es socio (si/no)?: ");
            String respuesta = teclado.next();
            if(respuesta.equalsIgnoreCase("si")){
                cuota -= CUOTA * PORCENTAJE_JOVEN_SOCIO / 100;
            }else{
                cuota -= CUOTA * PORCENTAJE_JOVEN_NO_SOCIO / 100;
            }
        }

        System.out.println("Debe pagar de cuota " + cuota + " Euros.");
    }
}
```

Comprueba si se ha escrito si en mayúsculas o minúsculas.

Ejercicio 4.10:

Escriba un método que calcule la potencia de un número real (a) elevado a un número entero (b). Tenga en cuenta que tanto a como b pueden valer 0 o pueden ser números negativos.

Planteamiento: La potencia de un número *a* elevado a otro *b*, es *a*a* … *a*, un número *b* de veces, si ambos son positivos. El resultado se complica si alguno o ambos valen 0 o un número es negativo. En particular, 0 elevado a un número negativo vale infinito. Para representar este valor se utilizará la constante `Double.POSITIVE_INFINITY`.

Solución:

```
public double potencia(double a, int b){
   double resultado = 1;
```

```
        if(a == 0){
            if (b == 0){
                resultado = 1;
            }else if(b < 0){
                resultado = Double.POSITIVE_INFINITY;
            }else{
                resultado = 0;
            }
        }else if(b == 0){
            resultado = 1;
        }else if (b < 0){
            for(int i = 0; i < -b; i++)
                resultado *= a;
            resultado = 1/resultado;
        }else{ // b es > 0
            for(int i = 0; i < b; i++)
                resultado *= a;
        }

        return resultado;
    }
```

0 elevado a un número negativo vale infinito.

Aviso: Puede comprobar el resultado de este método con lo que devuelve `Math.pow()`.

Ejercicio 4.11:

Escriba un programa, de nombre edad, para calcular la edad de una persona solicitando la fecha actual y la fecha de su nacimiento.

Planteamiento: El programa debe comenzar solicitando los datos de día, mes y año actuales y el día, mes y año en que nació la persona. Estos valores se guardarán en seis variables de tipo entero. En principio la edad de una persona se calcula restando del año actual el año de nacimiento. Pero si aún no ha sido su cumpleaños tendrá aún un año menos, por lo que si no ha sido aún su cumpleaños hay que restarle uno. Para comprobar que no ha sido su cumpleaños se comprueba si el mes actual es anterior al del cumpleaños o, si los meses coinciden, se comprueba si el día es anterior.

Solución:

```
import java.util.*;

public class EdadPersona {
    public static void main(String[] args){
        Scanner teclado = new Scanner(System.in);

        System.out.print("Introduzca el día de nacimiento: ");
        int díaNac = teclado.nextInt();
        System.out.print("Introduzca el mes de nacimiento: ");
        int mesNac = teclado.nextInt();
        System.out.print("Introduzca el año de nacimiento: ");
        int añoNac = teclado.nextInt();

        System.out.print("Introduzca el día actual: ");
        int díaHoy = teclado.nextInt();
```

```
        System.out.print("Introduzca el mes actual: ");
        int mesHoy = teclado.nextInt();
        System.out.print("Introduzca el año actual: ");
        int añoHoy = teclado.nextInt();

        int años = añoHoy - añoNac;
        if(mesHoy < mesNac){
            años--;
        }else if(mesHoy == mesNac && díaHoy < díaNac){   ← Si aún no ha cumplido los años.
            años--;
        }

        System.out.println("Tiene " + años + " años.");
    }
}
```

Comentario: El programa se podría haber hecho sin solicitar la fecha actual al usuario obteniendo este dato del sistema mediante el siguiente fragmento de código:

```
GregorianCalendar calendario = new GregorianCalendar();
int díaHoy = calendario.get(GregorianCalendar.DAY_OF_MONTH);
int mesHoy = calendario.get(GregorianCalendar.MONTH) + 1;
int añoHoy = calendario.get(GregorianCalendar.YEAR);
```

Ejercicio 4.12:

Escriba un método, de nombre esFechaVálida, que reciba como parámetros tres valores enteros con el día, mes y año de una fecha y devuelva un valor booleano que indique si se trata de valores válidos para una fecha.

Planteamiento: Los valores válidos para los meses son exclusivamente valores entre 1 y 12. Se considerarán como valores válidos para los años valores entre 1600 y 3000. En cuanto los valores válidos para los días dependerá del valor del mes de la fecha. Para comprobar si se trata de un día válido se utilizará una variable auxiliar en la que se le asigna el valor máximo de día del mes para el mes de la fecha. Mediante una estructura `switch` a esta variable se le asigna el valor 31 si es uno de los meses 1, 3, 5, 7, 8, 10 o 12. El valor 30 si es uno de los meses 4, 6, 9 u 11. En el caso de que el mes sea febrero, el 2, se debe tener en cuenta si se trata de un año bisiesto, para asignarle el valor 28 o 29. Un año es bisiesto si es múltiplo de cuatro y no es múltiplo de cien a no ser que sea múltiplo de cuatrocientos.

Parámetro: El método recibe por parámetro tres valores enteros, el valor del día del mes y del año de la fecha que se desea comprobar como válida.

Valor de retorno: El método devuelve un `boolean`, `true` si se trata de una fecha válida y `false` en caso contrario.

Solución:

```
public boolean esFechaVálida(int día, int mes, int año){
    int díasMes;

    // Se comprueba que el mes es correcto.
    if(mes < 1 || mes > 12)
        return false;
```

```
// Se comprueba que el año es correcto.
if(año < 1600 || año > 3000)
   return false;

// Es un mes correcto, se calcula los días que tiene.
   switch (mes){
      case 4:
      case 6:
      case 9:
      case 11:
         díasMes = 30;
         break;

      case 2:
         if ((año % 4 == 0) && (año % 100 != 0)
            || (año % 400 == 0)) {
            díasMes = 29;
         } else {
            díasMes = 28;
         }
         break;
      default:
         díasMes = 31;
         break;
   }

   // Se comprueba que el día es correcto.
   return día >= 1 && día <= díasMes;
}
```

Comentario: Esta forma de calcular los años bisiestos de una fecha sólo es correcta para el calendario gregoriano, es decir para fechas posteriores a 1582, fecha de su adopción.

Nota: Puede ver otro ejercicio de uso de switch en el Ejercicio 4.25.

ESTRUCTURAS DE REPETICIÓN

Ejercicio 4.13:

Escriba un método, de nombre esPrimo, que reciba un número entero por parámetro y devuelva un booleano que indique si se trata de un número primo o no.

Planteamiento: Para comprobar que se trata de un número primo hay que comprobar que no tiene divisores distintos de la unidad y del propio número. Se irá comprobando si tiene algún divisor a partir del 2 en adelante, si se encuentra algún número que lo divida, el método ejecutará una sentencia `return` con el valor `false`, pues si al menos tiene un divisor no es primo. Mediante un `for` se comprobará para todos los números entre el 2 y los menores que el valor pasado como argumento. Si al finalizar la ejecución del `for` en ningún momento se ha ejecutado la sentencia `return` será porque no tenía ningún divisor y se ejecutará la sentencia `return` con el valor `true`.

Parámetro: El método recibe como argumento el valor entero (`int`) del número del que se quiere comprobar que es primo. También se podría elegir un tipo `long`, ampliando así el rango de valores posibles.

Valor de retorno: El método devuelve un `boolean`, con el valor `true` si se trata de un número primo o `false` en caso de que no lo sea.

Solución:
```
public boolean esPrimo(int n){
   if (n % 2 == 0)
      return false;

   int divisor = 3;
   boolean primo = true;
   while (divisor * divisor <= n){
      if (n % divisor == 0)
            return false;
      divisor += 2;
   }
   return true;
}
```

Comentario: Recuerde que un número es primo si no tiene divisores distintos del propio número y la unidad. Primero se comprueba si es divisible por dos. En caso contrario, luego sólo se intenta dividir entre los impares a partir del 3.

Ejercicio 4.14:

*Escriba un programa que solicite al usuario un número entero positivo. El programa debe presentar en pantalla la descomposición en factores primos de dicho número. Por ejemplo, si el número es 28 debe escribir 36 = 2 * 2 * 3 * 3*

Planteamiento: Para ir calculando los sucesivos factores primos se va dividiendo el número por los sucesivos valores enteros empezando en dos. Cuando se encuentra un divisor se escribe y se actualiza el valor del número que se está factorizando hasta que es igual al propio divisor, en cuyo caso ese será el último factor primo.

Solución:
```
import java.util.*;
public class FactoresPrimos {
    public static void main(String[] args){
        long númeroInicial, número;
        Scanner teclado = new Scanner(System.in);

        System.out.print("Introduzca el número que desea factorizar: ");
        númeroInicial = teclado.nextLong();
        if(númeroInicial > 1){
            int factor = 2;
            número = númeroInicial;
            System.out.print(número + " = ");
            while (número > factor){
                if(número % factor == 0){
                    número /= factor;
                    System.out.print(factor + " * ");
                }else{
                    factor++;
                }
```

Si es divisible por factor, en la siguiente repetición hay que volver a dividir por factor si es posible.

```
            }
            System.out.println(factor);
        }
    }
}
```

Comentario: Idealmente habría que ir dividiendo sólo por los números primos, pero ello complica el ejercicio. En el ejemplo se divide por todos los números crecientemente, de esa forma si el factor no es primo, ya se habrá factorizado por uno de sus factores el número, por lo que en realidad nunca será divisible por un número no primo. Si el número es un primo grande o tiene un factor que es un primo grande, puede tardar bastante tiempo en realizar el cálculo completo.

Ejercicio 4.15:

Escriba un método que escriba en la pantalla el valor de la suma de los n primeros números pares.

Planteamiento: Para sumar los n primeros números pares hay que tener una variable que vaya acumulando dichos valores. Para ello se puede utilizar un bucle `for` que vaya generando para su índice los valores 0, 2, 4, ... , 2*(n-1) y los vaya sumando.

Solución:

```
import java.util.Scanner;
public class SumaPares {
    public static void main(String[] args) throws Exception{
        Scanner teclado = new Scanner(System.in);

        int número, suma=0;
        System.out.print("Introduzca el número n: ");
        número = teclado.nextInt();

        for(int i = 0; i < 2*número; i = i + 2){    // El índice va teniendo los valores 0, 2, 4, 6, ...
            suma = suma + i;
        }

        System.out.println("La suma de los primeros " + número +
                           " números pares es: " + suma);
    }
}
```

Comentario: En lugar de generar los números en el índice del `for` se podría generar los pares a partir del índice sustituyendo el bucle `for` del ejemplo por el siguiente:

```
for(int i = 0; i < número; i++){    // El índice va teniendo los valores 0, 1, 2, 3,... pero se suma 2*i
    suma = suma + 2 * i;
}
```

Ejercicio 4.16:

*Escriba un método dibujaCuadrado que dibuje un cuadrado de cuatro por cuatro mediante caracteres *, como el que se dibuja a continuación:*

```
* * * *
* * * *
* * * *
* * * *
```

Planteamiento: Se utiliza un bucle for para repetir cuatro veces el String "* * * *" cada vez en una línea.

Parámetros: No necesita ningún parámetro ya que el enunciado indica dibujar un cuadrado de 4x4.

Valor de retorno: El resultado se escribe directamente en pantalla. El método no necesita devolver ningún valor de retorno por lo que se utiliza la cláusula void.

Solución:
```
public void dibujaCuadrado(){
    for(int i = 0; i < 4; i++){
        System.out.println("* * * *");
    }
}
```

Comentario: Otra posible solución es utilizar dos for anidados, con un for interno que escriba cuatro asteriscos.

Ejercicio 4.17:

Escriba un método, de nombre dibujaRectangulo, que reciba dos valores enteros como parámetros, el método debe dibujar un rectángulo en el que la base es mayor que la altura, por lo que se utilizará como base el mayor de los valores proporcionados como parámetro y como altura el menor.

Por ejemplo si se le pasa como argumentos los valores 3 y 5 se dibujará el siguiente rectángulo:

```
* * * * *
* * * * *
* * * * *
```

Planteamiento: Después de comprobar cuál de los valores debe corresponder a la base y cuál a la altura del triángulo, se utiliza un for para repetir el número de filas con un for interior para repetir el número de asterisco por cada fila.

Parámetros: Tiene que recibir como parámetros el valor del ancho y alto del rectángulo como números enteros (int).

Valor de retorno: El resultado se escribe directamente en pantalla. El método no necesita devolver ningún valor de retorno por lo que se utiliza la cláusula void.

Solución:
```
public void dibujaRectangulo(int a, int b){

    int base   = (a > b) ? a : b; //Math.max(a, b);
    int altura = (a < b) ? a : b; //Math.min(a, b);
    for(int i = 0; i < altura; i++){
        for(int j = 0; j < base; j++){
            System.out.print("* ");
        }
        System.out.println();
    }
}
```

Comentario: Para decidir el valor correspondiente a la base y a la altura se pueden utilizar los métodos max() y min() de la clase Math, de la forma indicada en el comentario.

Ejercicio 4.18:

Escriba un método, de nombre dibujaTriángulo, que reciba un entero como parámetros, el método debe dibujar un triángulo como el que se muestra en el siguiente dibujo después de comprobar que el valor del parámetro es menor de 15.

Por ejemplo si el valor del argumento es 4 el método dibujará el siguiente triángulo:

```
* * * *
* * *
* *
*
```

Planteamiento: Si el valor del argumento es menor que 15 se utilizan dos bucles `for` anidados para dibujar el triángulo, para que en cada una de las filas escriba un asterisco menos, en el bucle `for` interior se ejecuta un número de veces distinto en cada fila.

Parámetros: Tiene que recibir como parámetro el valor del lado del triángulo como un número entero (`int`).

Valor de retorno: El resultado se escribe directamente en pantalla. El método no necesita devolver ningún valor de retorno por lo que se utiliza la cláusula `void`.

Solución:

```
public void dibujaTriángulo(int lado){
    if (lado < 15){
        for(int i = 0; i < lado; i++){
            for(int j = 0; j < lado-i; j++){
                System.out.print("* ");
            }
            System.out.println();
        }
    }else{
        System.out.println("El lado es demasiado grande para dibujar el triángulo.");
    }
}
```

Ejercicio 4.19:

Escriba un método, de nombre dibujaTriángulo2, que reciba un entero como parámetro, el método debe dibujar un triángulo como el que se muestra en el siguiente dibujo después de comprobar que el valor del parámetro se encuentra entre 3 y 15, ambos incluidos.

Por ejemplo si el valor del argumento es 4 el método dibujará el siguiente triángulo:

```
      *
    * *
  * * *
* * * *
```

Planteamiento: Se comprueba que el valor del argumento es válido, para dibujar este triángulo se comienzan por escribir mediante un `for` los caracteres en blanco correspondientes a cada fila y a continuación mediante otro `for` los asteriscos correspondientes.

Parámetros: Tiene que recibir como parámetro el valor del lado del triángulo como un número entero (`int`).

Valor de retorno: El resultado se escribe directamente en pantalla. El método no necesita devolver ningún valor de retorno por lo que se utiliza la cláusula `void`.

Solución:
```
public void dibujaTriángulo2(int lado){
    if (lado > 3 && lado < 15){
        for(int i = 0; i < lado; i++){
            for(int j = 0; j < lado-i-1; j++){
                System.out.print("  ");
            }
            for(int j = 0; j < i+1; j++){
                System.out.print("* ");
            }
            System.out.println();
        }
    }else{
        System.out.println("El lado del triangulo debe estar entre 3 y 15.");
    }
}
```

Ejercicio 4.20:

Escriba un método, de nombre dibujaTriángulo3, que reciba un entero como parámetro, el método debe dibujar un triángulo como el que se muestra en el siguiente dibujo después de comprobar que el valor del parámetro es un número impar positivo.

Por ejemplo si el valor del parámetro es 7 el método dibujará el siguiente triángulo con 7 asteriscos en la base:

```
      *
    * * *
  * * * * *
* * * * * * *
```

Planteamiento: Se utiliza un bucle `for` para escribir tantos espacios como sean necesarios en cada fila y luego otro bucle `for` para escribir tantos asteriscos como sean necesarios en cada fila.

Parámetros: Tiene que recibir como parámetro el valor del número de asteriscos en base del triángulo como un número entero (`int`).

Valor de retorno: El resultado se escribe directamente en pantalla. El método no necesita devolver ningún valor de retorno por lo que se utiliza la cláusula `void`.

Solución:
```
public void dibujaTriángulo3(int lado){
   if (lado % 2 == 1 && lado > 0){
      for(int i = 0; i < (lado + 1)/2; i++){
         for(int j = 0; j < (lado + 1)/2 - i-1; j++){
            System.out.print("  ");
         }
         for(int j = 0; j < 2*i + 1; j++){
            System.out.print("* ");
         }
```

```
                System.out.println();
            }
        }else{
            System.out.println("El lado del triangulo debe ser un impar positivo.");
        }
    }
```

Comentario: Como puede observar se ha utilizado una fórmula que calcula cuántos espacios y cuántos asteriscos tiene que imprimir dada la fila correspondiente.

Una forma que puede resultar más sencilla es pensar que en la primera fila hay que imprimir tantos espacios como valga el lado/2, y en las siguientes siempre uno más cada vez. En cuanto a los asteriscos hay que imprimir uno en la primera fila y en las demás dos más por cada fila.

De esta forma la solución sería:

```
public void dibujaTriángulo3(int lado) {
    int espacios = lado / 2;
    int asteriscos = 1;
    if (lado % 2 == 1 && lado > 0){
        for(int i = 0; i < (lado + 1)/2; i++){
            for(int j = 0; j < espacios; j++){
                System.out.print("  ");
            }
            for(int j = 0; j < asteriscos; j++){
                System.out.print("* ");
            }
            System.out.println();

            espacios   -= 1;
            asteriscos += 2;
        }
    }else{
        System.out.println("El lado del triángulo debe ser un impar positivo.");
    }
}
```

Cada fila siguiente tiene un espacio menos y dos asteriscos más.

Ejercicio 4.21:

Escriba un método dibujaRombo que reciba un entero como parámetro. El método debe dibujar un rombo como el que se muestra en el siguiente dibujo después de comprobar que el valor que recibe como argumento, que indica la altura del rombo, es un número impar mayor o igual que 3.

Por ejemplo si el valor del parámetro es 5 el método dibujará el siguiente rombo con 5 asteriscos en la parte central:

```
    *
  * * *
* * * * *
  * * *
    *
```

Planteamiento: La solución es muy parecida a la del Ejercicio 4.20. Se irán escribiendo tantos espacios y asteriscos en cada fila como sean necesarios dependiendo de la fila. Esto se hace para la mitad superior del rombo y luego para la mitad inferior.

Parámetros: Tiene que recibir como parámetro el valor del número de asteriscos en la parte central del rombo como un número entero (`int`).

Valor de retorno: El resultado se escribe directamente en pantalla. El método no necesita devolver ningún valor de retorno por lo que se utiliza la cláusula `void`.

Solución:

```
public void dibujaRombo(int altura){
    if (altura % 2 == 1 && altura >= 3){
        // se dibuja la parte superior
        for(int i = 0; i < (altura + 1)/2; i++){
            for(int j = 0; j < (altura + 1)/2 - i-1; j++){
                System.out.print("  ");
            }
            for(int j = 0; j < 2*i + 1; j++){
                System.out.print("* ");
            }
            System.out.println();
        }

        // se dibuja la parte inferior
        for(int i = 0; i < altura/2; i++){
            for(int j = 0; j < i + 1; j++){
                System.out.print("  ");
            }
            for(int j = 0; j < (altura/2 - i)*2 - 1; j++){
                System.out.print("* ");
            }
            System.out.println();
        }
    }else{
        System.out.println("El lado del triángulo debe ser un impar positivo.");
    }
}
```

Expresión que calcula cuántos espacios hay que escribir en la fila i.

Expresión que calcula cuántos asteriscos hay que escribir en la fila i de la parte inferior.

Comentario: Igual que se ha hecho en el Ejercicio 4.20, podría hacerse también llevando la cuenta del número de espacios y asteriscos que hay que ir escribiendo en cada fila, de la siguiente forma:

```
public void dibujaRombo(int altura){
    if (altura % 2 == 1 && altura >= 3){
        int espacios = altura / 2;
        int asteriscos = 1;

        // se dibuja la parte superior
        for(int i = 0; i < (altura + 1)/2; i++){
            for(int j = 0; j < espacios; j++){
                System.out.print("  ");
            }
            for(int j = 0; j < asteriscos; j++){
                System.out.print("* ");
            }
```

```
            System.out.println();
            espacios   -= 1;
            asteriscos += 2;
        }

        espacios = 1;
        asteriscos = altura - 2;
        // se dibuja la parte inferior
        for(int i = 0; i < altura/2; i++){
            for(int j = 0; j < espacios; j++){
                System.out.print("  ");
            }
            for(int j = 0; j < asteriscos; j++){
                System.out.print("* ");
            }
            System.out.println();
            espacios   += 1;
            asteriscos -= 2;
        }
    }else{
        System.out.println("El lado del triángulo debe ser un impar positivo.");
    }
}
```

Cada fila siguiente tiene un espacio menos y dos asteriscos más.

Cada fila siguiente tiene un espacio más y dos asteriscos menos.

Ejercicio 4.22:

Escriba un método escribeAlRevés que reciba un String y escriba dicho texto en la pantalla al revés.

Planteamiento: Para escribir el texto al revés se puede ir escribiendo el texto desde el último carácter del String hasta el primero de uno en uno.

Parámetros: Tiene como parámetro el texto que se desea escribir al revés (String).

Valor de retorno: El resultado se escribe directamente en pantalla. El método no necesita devolver ningún valor de retorno por lo que se utiliza la cláusula void.

Solución:

```
public void escribeAlRevés(String texto){
    for(int i = texto.length()-1; i >= 0; i-){
        System.out.print(texto.charAt(i));
    }
}
```

Aviso: Habría que comprobar al principio qué texto no vale null.

Ejercicio 4.23:

Escriba un método que compruebe si una palabra o una frase es un palíndromo.

Planteamiento: Para comprobar que una frase es un palíndromo hay que comprobar si se corresponden el carácter de su izquierda con el de su derecha y así uno a uno hasta llegar a la mitad. Para no tener en cuenta si los caracteres están en mayúsculas o en minúsculas antes de hacer la comprobación se deben convertir los caracteres bien a mayúsculas, bien a minúsculas. Se va a escribir la solución convirtiéndolos a mayúsculas.

Parámetros: Tiene como parámetro el texto que se desea comprobar (`String`).

Valor de retorno: Debe devolver si el texto es realmente un palíndromo o no, por tanto debe devolver un `boolean`.

Solución:

```
public boolean palindromo(String frase){
    for(int i = 0; i < frase.length() / 2; i++){
        if(Character.toUpperCase(frase.charAt(i)) != ←
          Character.toUpperCase(frase.charAt(frase.length() - 1 - i)))
            return false;
    }
    return true;
}
```

Compara los caracteres como mayúsculas.

Comentario: Otra solución podría ser construir otra cadena que sea la que el método recibe como argumento pero construyéndola al revés y compararla con la original. Para realizar la comparación sin tener en cuenta mayúsculas y minúsculas se puede utilizar el método `equalsIgnoreCase()`.

Aviso: Habría que comprobar al principio qué texto no vale `null`.

Ejercicio 4.24:

Escriba un método que reciba como parámetro un String y devuelva el número de palabras que contiene el String. Se consideran palabras los grupos de caracteres separados por espacios en blanco.

Planteamiento: Para devolver el número de palabras que tiene el `String` hay que ir recorriendo el array e ir contando el número de huecos con espacios que hay entre las palabras. Si se supone que entre una palabra y la siguiente sólo hay un espacio se puede escribir la siguiente solución

Parámetros: Tiene como parámetro el texto del que se desea contar las palabras (`String`).

Valor de retorno: Debe devolver el número de palabras del texto, que siempre es un valor entero, es decir un valor del tipo `int`.

Solución:

```
public int cuentaPalabras(String texto){
   int palabras = 0;
   String textoAux = texto.trim(); ←
   if(textoAux.length() == 0)
       return 0;

   for(int i = 0; i < textoAux.length(); i++){
       if(textoAux.charAt(i) == ' ') ←
           palabras++;
   }
   return palabras + 1;
}
```

El método `trim()` elimina los espacios que haya al inicio y final del texto.

En realidad cuenta el número de espacios.

Comentario: El método anterior es la forma sencilla. Sin embargo, en general entre dos palabras habrá un número cualquiera de espacios en blanco, de forma que cuando se encuentre uno hay que saltar todos los que vayan a continuación.

```
public static int cuentaPalabras3(String texto){
    int palabras = 0;
    String textoAux = texto.trim();
    if(textoAux.length() == 0)
        return 0;

    for(int i = 0; i < textoAux.length(); i++){
        if(textoAux.charAt(i) == ' '){
          while(i < textoAux.length() && textoAux.charAt(i) == " ")
                i++;
          palabras++;
        }
    }
    return palabras + 1;
}
```

Cuando encuentra un espacio salta todos los espacios que haya a continuación.

Comentario: Otra forma es considerar que hay que saltar por una parte los espacios en blanco y, por otra parte, los caracteres que no son espacios en blanco.

```
public int cuentaPalabras(String texto){
    int palabras = 0, i = 0;
    String textoAux = texto.trim();
    while(true){
        if(i == textoAux.length())
            break;
        while(i < textoAux.length() && textoAux.charAt(i) != " ")
            i++;
        palabras++;
        while(i < textoAux.length() && textoAux.charAt(i) == " ")
            i++;
    }
    return palabras;
}
```

Comentario: La forma más sencilla de escribir este método es utilizar el método `split()`, como si hizo en el Ejercicio 3.19, que devuelve un array de `String`, dividiendo el que se pasa como parámetro utilizando una expresión regular (consulte la clase `Pattern`). Se utiliza como expresión regular la expresión " +" que significa un espacio una o más veces. Pero `split()` crea un array que luego no se utiliza.

```
public int cuentaPalabras(String texto){
   return texto.trim().split(" +").length;
}
```

Aviso: Habría que comprobar al principio qué texto no vale `null`.

Ejercicio 4.25:

Escriba un programa que cuente el número de vocales que aparecen en un texto que se solicita al usuario.

Planteamiento: Hay que solicitar al usuario un texto. Una vez obtenido hay que recorrer el texto carácter a carácter e ir comprobando si cada uno de ellos se corresponde con una vocal. Si es así, se cuenta la vocal a la que corresponde.

Solucion:

```
import java.util.*;

public class Vocales {
    public static void main(String[] args){
        int vocalA, vocalE, vocalI, vocalO, vocalU;
        vocalA = vocalE = vocalI = vocalO = vocalU = 0;   <-- Inicializa todas las variables a 0.

        Scanner teclado = new Scanner(System.in);

        System.out.print("Introduzca un texto: ");
        String texto = teclado.nextLine();
        for(int i = 0; i < texto.length(); i++){
            switch(Character.toUpperCase(texto.charAt(i))){   <-- Se convierte la letra a mayúscula y sólo hay que comprobar si coincide con la mayúscula correspondiente.
                case 'A':
                    vocalA++;
                    break;
                case 'E':
                    vocalE++;
                    break;
                case 'I':
                    vocalI++;
                    break;
                case 'O':
                    vocalO++;
                    break;
                case 'U':
                    vocalU++;
                    break;
                default:
                    break;
            }
        }

        System.out.println("La vocal A aparece " + vocalA + " veces.");
        System.out.println("La vocal E aparece " + vocalE + " veces.");
        System.out.println("La vocal I aparece " + vocalI + " veces.");
        System.out.println("La vocal O aparece " + vocalO + " veces.");
        System.out.println("La vocal U aparece " + vocalU + " veces.");
    }
}
```

Ejercicio 4.26:

Escriba un método que permita realizar el cifrado César de un texto. Para realizar el cifrado César se necesita el texto a cifrar y un número que indica el desplazamiento que se emplea de los caracteres para cifrar. Si el número es 2 la letra 'a' se sustituye por la letra 'c', la 'b' por la 'd', etc. Utilice para el cifrado el alfabeto español que incluya la ñ. Para simplificar el ejercicio suponga que los textos son en minúsculas. Los caracteres que no pertenezcan al alfabeto en minúsculas permanecerán como en el original.

Planteamiento: Para hacer el cifrado César se necesita disponer del alfabeto completo. Para ello se puede utilizar un `String` que contenga todos los caracteres a cifrar en orden, incluyendo la ñ en su posición. Para

realizar el cifrado se buscará si la letra a cifrar se encuentra en el alfabeto y si es así en qué posición está. Para cifrarla basta sumarle el número dado de cifrado. Si al sumar se excede del tamaño del alfabeto se puede volver al principio del mismo utilizando la función %.

Parámetros: Tiene como parámetro el texto que se desea cifrar (String) y el número que se usa para el cifrado (int).

Valor de retorno: Debe devolver el texto cifrado, es decir, un objeto de la clase String.

Solución:

```
public String cifrar(String texto, int número){
    final String alfabeto = "abcdefghijklmnñopqrstuvwxyz";

    String cifrado = "";
    for(int i = 0; i < texto.length(); i++){
        int pos = alfabeto.indexOf(texto.charAt(i));
        if(pos >= 0)
            cifrado += alfabeto.charAt((pos + número) % alfabeto.length());
        else
            cifrado += texto.charAt(i);
    }
    return cifrado;
}
```

Se utiliza la operación módulo para volver al inicio del alfabeto cuando se sale por el final.

Aviso: Habría que comprobar al principio qué texto no vale null. Sin embargo, no es necesario comprobar el valor de la variable entera número. Si dicho valor es negativo cifra en sentido contrario, lo que permite descifrar un texto. Si el valor es 0, en realidad el texto no se modifica.

Ejercicio 4.27:

Escriba un programa que permita determinar si utilizar los números aleatorios de la clase Math son apropiados. Para ello el programa debe simular que se lanza una moneda un número elevado de veces, por ejemplo, 1.000.000. A continuación debe imprimir por pantalla el porcentaje de caras y el porcentaje de cruces que han salido.

Planteamiento: Se va a utilizar el método Math.random() que devuelve un valor entre 0 y 1, excluido el último. Para simular la tirada de una moneda se puede multiplicar por 2 este valor y comprobar si el resultado es menor que 1 o no.

Solución:

```
public class TirarMoneda{
    public static void main(String[] args){
        int veces = 1000000;
        int caras = 0;

        for(int i = 0; i < veces; i++){
            if(Math.random()*2 < 1)
                caras++;
        }
        System.out.printf("Caras  %.2f%%\n", caras*100.0/veces);
        System.out.printf("Cruces %.2f%%\n", (veces-caras)*100.0/veces);
    }
}
```

Comentario: También se puede hacer comprobando si el resultado de llamar a Math.random() es menor que 0.5.

Aviso: Si desea un buen generador de números aleatorios utilice objetos de la clase Random.

Ejercicio 4.28:

Escriba un programa que juegue con el usuario a adivinar un número. El ordenador debe generar un número entre 1 y 100 y el usuario tiene que intentar adivinarlo. Para ello, cada vez que el usuario introduce un valor el ordenador debe decirle al usuario si el número que tiene que adivinar es mayor o menor que el que ha introducido. Cuando consiga adivinarlo debe indicárselo e imprimir en pantalla el número de veces que el usuario ha intentado adivinar el número. Si el usuario introduce algo que no es un número debe indicarlo de esta forma en pantalla y contarlo como un intento.

Planteamiento: Para realizar este juego se guarda en una variable (númeroAdivinar) un número aleatorio entre 1 y 100. Se solicita al usuario un número mientras no haya acertado. Como al menos hay que pedirlo una vez, el bucle más apropiado es un bucle do-while.

Solución:

```
import java.util.*;
public class AdivinaNúmero {
    public static void main (String []args){
        Scanner teclado = new Scanner(System.in);

        int númeroAdivinar = (int)(Math.random()*100);
        // debe ser entre 1 y 100, por lo que se incrementa en uno
        númeroAdivinar++;
        int número = 0;
        int intentos = 0;
        do{
            System.out.print("Introduzca un número (1-100): ");
            número = teclado.nextInt();
            intentos++;
            System.out.println("El número introducido fue: " + número);

            if (número > númeroAdivinar)
                System.out.println("El número es menor. Inténtalo otra vez.");

            if (número < númeroAdivinar)
                System.out.println("El número es mayor. Inténtalo otra vez.");
        }while(número != númeroAdivinar);
        System.out.println("Enhorabuena. Ya lo ha adivinado. El número era: " + númeroAdivinar);
        System.out.println("Lo ha conseguido en " + intentos + " intentos.");
    }
}
```

Genera un número entre 0 y 99 y lo trunca a entero.

Aviso: Tenga en cuenta que si introduce algo que no es un número entero el programa termina con una excepción. Sobre el manejo de excepciones vea ejercicios más adelante en este mismo capítulo.

Ejercicio 4.29:

Escriba un programa que permita determinar la probabilidad con que aparece cada uno de los valores al lanzar un dado. Para ello se lanzará el dado 1.000.000 de veces y se imprimirá por pantalla cuántas veces ha aparecido cada número y el porcentaje que representa respecto al total.

Planteamiento: Para modelar el lanzamiento de un dado se utiliza el método `Math.random()` que devuelve un valor entre 0 y 1, excluido el último. Para simular la tirada de un dado se multiplicará el valor por 6 y se tomará la parte entera (de 0 a 5).

Solución:
```
public class Dados {
    public static void main(String[] args){
        double valorAleatorio;
        int valorDado;
        int numTiradas = 1000000;
        int[] tiradas = new int[6];

        // se lanza el dado 1.000.000 veces
        for(int i=0; i < numTiradas; i++){
            valorAleatorio = Math.random()*6;
            valorDado = (int)(valorAleatorio + 1);  <--- Para que el número generado esté entre 1 y 6, como un dado.

            tiradas[valorDado-1]++;
        }

        // se imprime cuántas veces ha aparecido cada número y su porcentaje
        for(int i = 0; i < tiradas.length; i++){
            System.out.printf("El valor %d aparece %d lo que representa un %.2f%%.\n",
                    i + 1, tiradas[i], (tiradas[i] / (double)numTiradas) * 100);
        }
    }
}
```

Aviso: El uso de arrays se verá en el Capítulo 6.

EXCEPCIONES

Ejercicio 4.30:

Escriba un programa que juegue con el usuario a adivinar un número. El ordenador debe generar un número entre 1 y 100 y el usuario tiene que intentar adivinarlo. Para ello, cada vez que el usuario introduce un valor el ordenador debe decirle al usuario si el número que tiene que adivinar es mayor o menor que el que ha introducido. Cuando consiga adivinarlo debe indicárselo e imprimir en pantalla el número de veces que el usuario ha intentado adivinar el número. Si el usuario introduce algo que no es un número debe indicarlo de esta forma en pantalla y contarlo como un intento.

Planteamiento: Para realizar este juego se guarda en una variable (`númeroAdivinar`) un número aleatorio entre 1 y 100. Se solicita al usuario un número mientras no haya acertado. Como al menos hay que pedirlo una vez el bucle más apropiado es un bucle `do-while`. El ejercicio es como el Ejercicio 4.28, pero hay que controlar cada lectura de dato. Para ello, hace falta un segundo bucle `do-while` dentro del anterior para controlar que la lectura que se realiza es realmente un número entero y en caso contrario dar un mensaje de error y volver a pedirlo. Scanner indica que no ha conseguido reconocer la entrada lanzando la excepción `InputMismatchException`. Para saber si cada vez que se pide se ha conseguido leer un número entero se utiliza una variable `boolean` que indique si se ha leído correctamente o no.

Solución:
```
import java.util.*;
```

```
public class AdivinaNúmero {
    public static void main (String []args){
        Scanner teclado = new Scanner(System.in);

        int númeroAdivinar = (int)(Math.random()*100);
        // debe ser entre 1 y 100, por lo que se incrementa en uno
        númeroAdivinar++;
        int número = 0;
        int intentos = 0;
        do{
            boolean leído;
            do{
                System.out.print("Introduzca un número (1-100): ");
                try{
                    intentos++;
                    número = teclado.nextInt();
                    System.out.println("El número introducido fue: " + número);
                    leído = true;
                }catch(InputMismatchException e){
                    System.out.println("No ha introducido un número entero.");
                    teclado.next();
                    leído = false;
                }
            }while(leído == false);

            if (número > númeroAdivinar)
                System.out.println("El número es menor. Inténtalo otra vez.");

            if (número < númeroAdivinar)
                System.out.println("El número es mayor. Inténtalo otra vez.");

        }while(número != númeroAdivinar);

        System.out.println("Enhorabuena, lo has adivinado. El número era: " + númeroAdivinar);
        System.out.println("Lo has conseguido en " + intentos + " intentos.");
    }
}
```

Se lee el elemento no reconocido para eliminarlo.

Comentario: Hubiese resultado más natural escribir el bucle `do-while` que comprueba que se lea realmente un número entero de la siguiente forma:

```
do{
   // solicitar un número al usuario
}while(!leído);
```

Ejercicio 4.31:

Escriba un programa que lance y capture una excepción de la clase Exception

Planteamiento: El programa abre un bucle `try{}` en el que comienza mostrando un mensaje por pantalla. A continuación, crea un objeto de la clase `Exception` indicando en su constructor un mensaje explicativo. Se procede a continuación a su lanzamiento utilizando la sentencia `throw`. El bloque `catch{}` asociado al bloque

try{} anterior constituye el manejador de la excepción e incluye las sentencias necesarias para su tratamiento. En este caso el manejado se limita a mostrar por pantalla el mensaje contenido en el objeto excepción capturado que es obtenido con ayuda del método getMessage().

Solución:

```
public static void main(String args[]){
    try{
        System.out.println("Mensaje mostrado por pantalla");
        Exception e = new Exception("Esto es un objeto Exception");
        throw e;
    }catch(Exception e1){
        System.out.println("Excepcion capturada con mensaje: "+e1.getMessage());
    }
    System.out.println("Programa terminado.");
}
```

Construcción del objeto que representa la excepción.

Lanzamiento de la excepción.

Manejador de la excepción.

Comentario: Después del bloque catch() se ha introducido una sentencia para mostrar un mensaje de final de programa. Con ello se pretende ilustrar que el flujo de ejecución continúa secuencialmente después de dicho bloque. Si se tratara de incluir una sentencia a continuación de la sentencia throw, por ejemplo, para intentar mostrar un nuevo mensaje por pantalla, se obtendría el siguiente error de compilación.

```
C:\ EjExceptions\src\ejexceptions\EjExceptionsBasicos.java:31: unreachable statement
            System.out.println("Mensaje también mostrado por pantalla");
```

Esto ocurre porque la sentencia throw provocará que el flujo de ejecución salte al inicio del código establecido como manejador de la excepción, en este caso la primera sentencia del bloque catch(). Por consiguiente, cualquier sentencia incluida a continuación de throw no podrá ser ejecutada nunca (unreachable statement).

Ejercicio 4.32:

Escriba un programa que genere un número aleatorio e indique si el número generado es par o impar. El programa utilizará para ello el lanzamiento de una excepción.

Planteamiento: El programa utiliza el método random() de la clase Math para obtener un número aleatorio entre 0 y 99. A continuación, se determina si el número es par o impar y se lanza una excepción con el correspondiente mensaje para indicarlo. El manejador de la excepción se limitará a mostrar el mensaje asociado a la excepción capturada.

Solución:

```
public static void main(String args[]){
    try{
        System.out.println("Generando numero aleatorio.... ");
        int entero = (int)(Math.random() * 100);
        if(entero%2 == 0){
            throw new Exception("Se generó el numero par: " + entero);
        }else{
            throw new Exception("Se generó el numero impar: " + entero);
        }
    }
    catch(Exception e){
        System.out.println(e.getMessage());
    }
}
```

Generación de un número aleatorio entre 0 y 99.

El objeto excepción se construye y se lanza en la misma sentencia.

Comentario: El método `random()` devuelve un número real (`double`) entre 0 (incluido) y 1(no incluido). Este valor es multiplicado por 100 y convertido a entero para obtener un valor entre 0 y 99 (ambos incluidos). Por supuesto, este programa podría haberse implementado de una forma más sencilla incluyendo una sentencia de bifurcación o el operador ternario para mostrar directamente si el número generado es par o impar. Se ha realizado de la forma presentada para aportar un nuevo ejemplo de utilización de excepciones.

```
public static void main(String args[]){
   System.out.println("Generando número aleatorio.... ");
   int entero = (int)(Math.random() * 100);
   System.out.println("Se generó el número " +
                     (entero%2 == 0 ? "par " : "impar") +
                     ": " + entero);
 }
```

Ejercicio 4.33:

Escriba un programa que genere números aleatorios entre 0 y 99 hasta obtener 5 números impares. Los números impares se almacenarán en un array y se mostrarán al final del programa. El programa también indicará el número de valores aleatorios que ha sido necesario generar hasta obtener los 5 valores impares.

Planteamiento: El programa declara un array donde se almacenarán los 5 valores impares obtenidos. Se utiliza un bucle infinito donde se irán generando valores aleatorios. Cuando el valor corresponda a un número impar se incluirá en el array, incrementando una variable índice para la inserción del siguiente elemento. Cuando se hayan obtenido 5 valores impares, o lo que es lo mismo, cuando la variable índice alcance la longitud del array, se lanzará una excepción para notificar la situación que incluirá un mensaje con el número de valores aleatorios que ha sido necesario generar. El lanzamiento de la excepción provocará un salto a la primera sentencia del manejador de la excepción, interrumpiendo así la ejecución del bucle infinito. El manejador de la excepción mostrará el mensaje del objeto excepción y, a continuación, los impares generados aleatoriamente.

Solución:

```
public static void main(String args[]){
    int[] impares= new int[5];
    try{
        int intentos=0,i=0;
        while(true){
            int numero=(int)(Math.random() * 100);
            intentos++;
            if(numero % 2 != 0){
                impares[i++] = numero;
            }
            if(i == 5){
                throw new Exception("Se generaron los 5 números impares en " +
                                    intentos + " intentos");
            }
        }
    }catch(Exception e){
        System.out.println(e.getMessage());
        System.out.println("Los impares generados son: " + Arrays.toString(impares));
    }
}
```

Generación de un número aleatorio entre 0 y 99.

Excepción lanzada cuando se han obtenido 5 valores impares.

Comentario: Para mostrar el contenido del array sin necesidad de utilizar un bucle para acceder a sus posiciones se utiliza el método toString() de la clase de utilidades Arrays. Este método se explicará con más detalle en el Capítulo 6 sobre Estructuras de almacenamiento.

Ejercicio 4.34:

Escriba un programa que lea el primer carácter almacenado en un archivo de nombre "C:/misArchivos/alfabeto.txt" y, a continuación, lo muestre por pantalla.

Planteamiento: El programa trata de abrir un flujo de lectura sobre el fichero solicitado. A continuación, el programa utiliza el método read() para obtener el primer carácter del citado archivo. El intento de apertura de un flujo de lectura sobre un archivo puede generar una excepción de la clase FileNotFoundException si el fichero no existe o no tiene permiso de lectura. Por su parte, la lectura del primer carácter puede ocasionar una excepción de la clase IOException si ocurre algún error de lectura sobre el flujo. Para notificar convenientemente uno u otro error, se capturan y procesan las excepciones por separado.

Solución:

```
public static void main(String args[]){
    FileReader lectura;
    try{
        lectura = new FileReader("C:/misArchivos/alfabeto.txt");
        char pcar=(char)lectura.read();
        System.out.println("Primer carácter: " + pcar);
    }
    catch(FileNotFoundException fnfE){
        System.out.println("El archivo no se pudo abrir para lectura");
    }
    catch(IOException ioE){
        System.out.println("Error de lectura");
    }
    finally{
        try{
            lectura.close();
        }catch(IOException ioE){}
    }
}
```

Captura de las excepciones de tipo FileNotFoundException.

Captura de las excepciones de tipo IOException.

Se supone que no se lanza. Si se lanza se ignora.

Comentario: Dado que la clase FileNotFoundException deriva de IOException su manejador debe ser incluido antes. Cuando se genera una excepción se recorren todos los bloques catch{} hasta que alguno de ellos pueda procesar la excepción. Un bloque catch{} actuará como manejador de la excepción que puede capturar y también de todas sus excepciones derivadas. Por esta razón, si se escribiera primero el bloque catch{} de IOException se procedería también a tratar en él la excepción de tipo FileNotFoundException. Si el bloque correspondiente a la última excepción se escribe después, se obtendrá un error de compilación como el siguiente:

```
C:\ EjExceptions\src\ejexceptions\EjExceptionsBasicos.java:128: exception
java.io.FileNotFoundException has already been caught)
```

ya que, efectivamente, cualquier excepción de la clase FileNotFoundException será capturada en el bloque previo. Por último el bloque finally siempre se ejecuta por lo que, ocurra lo que ocurra, se cerrará el archivo.

Ejercicio 4.35:

Escriba un programa que lea el primer carácter de los archivos C:/misArchivos/archivo1 y /archivo2.

Planteamiento: El método utiliza dos bloques try{}catch{} para crear un flujo de lectura y obtener el primer carácter de cada uno de los archivos. En los dos bloques catch{} sólo se captura la correspondiente excepción que se lanzaría si alguno de los dos archivos no estuviera disponible para su lectura. Ninguno de los dos bloques catch{} captura la excepción que podría derivarse de la operación de lectura sobre los archivos, cuyo tratamiento se deja a un bloque catch{} más externo. De esta forma se consigue tratar de una forma individual el posible error en el momento de creación de un flujo de escritura sobre cada archivo y de forma conjunta la ocurrencia de un error de lectura.

Solución:

```
public static void main(String args[]){
   FileReader lectura;
   try{
       try{
            lectura = new FileReader("C:/misArchivos/archivo1.txt");
            char pcar=(char)lectura.read();
            System.out.println("Primer caracter archivo1.txt: " + pcar);
        }
        catch(FileNotFoundException fnfE){
            System.out.println(
             "El archivo C:/misArchivos/archivo1.txt no se pudo abrir para lectura");
        }
        finally{
            try{
                lectura.close();
            }catch(IOException ioE){}
        }
        try{
            lectura = new FileReader("C:/misArchivos/archivo2.txt");
            char pcar=(char)lectura.read();
            System.out.println("Primer caracter archivo2.txt: " + pcar);
        }
        catch(FileNotFoundException fnfE){
            System.out.println(
             "El archivo C:/misArchivos/archivo2.txt no se pudo abrir para lectura");
        }
        finally{
            try{
                lectura.close();
            }catch(IOException ioE){}
        }
   }catch(IOException ioE){
       System.out.println("Error de lectura en algún archivo");
   }
}
```

Captura de las excepciones de tipo IOException.

Comentario: Cuando se lanza una excepción dentro de un bloque try{}catch{} ésta tratará de capturarse en su bloque catch{} asociado. Si dicho bloque no ha definido un manejador para la excepción su captura tratará de realizarse en el bloque catch{} inmediatamente más externo. El flujo de ejecución saltará, así, a la primera sentencia del manejador más externo definido para la excepción. El ejercicio podría haberse resuelto utilizando un único bloque try{}catch{} para la apertura de los dos flujos de lectura sobre ambos archivos, pero si la apertura del primer flujo da lugar a un error, el flujo de control saltará al manejador con lo que el segundo flujo tampoco sería abierto.

```
public static void main(String args[]){
    FileReader lectura1;
    FileReader lectura2;
    try{
        lectura1 = new FileReader("C:/misArchivos/archivo1.txt");
        char pcar=(char)lectura1.read();
        System.out.println("Primer carácter archivo1.txt: " + pcar);
        lectura2 = new FileReader("C:/misArchivos/archivo2.txt");
        pcar=(char)lectura2.read();
        System.out.println("Primer carácter archivo2.txt: " + pcar);
     }
     catch(FileNotFoundException fnfE){
         System.out.println(
             "El archivo C:/misArchivos/archivo1.txt no se pudo abrir para lectura");
     }
     catch(IOException ioE){
           System.out.println("Error de lectura en algún archivo");
     }
     finally{
         try{
             lectura1.close();
             lectura2.close();
         }catch(IOException ioE){}
     }
}
```

Captura de las excepciones de tipo IOException.

Ejercicio 4.36:

Indique la salida por pantalla que produciría el siguiente programa:

```
public class EjemploExcepciones {

    public static int devuelveEntero(int num){
        try{
            if(num % 2 == 0){
             throw new Exception("Lanzando excepción");
            }
            return 1;
        }
        catch(Exception e){
            return 2;
        }
        finally{
            return 3;
        }
    }
    public static void main(String args[]){
        System.out.println(devuelveEntero(1));
    }
}
```

¿Y si la llamada al método devuelveEntero se realiza con el valor 2?

Solución: En ambos casos se mostrará por pantalla el valor 3, ya que es el que devuelve el bloque `finally{}`. Este bloque es opcional, pero si se incluye, sus sentencias se ejecutan siempre.

Ejercicio 4.37:

Considere el siguiente programa:

```
public class EjemploExcepciones2 {

    public static int devuelveEntero(int num) throws IOException{
        try{
            if(num % 2 == 0){
             throw new Exception("Lanzando excepción");
            }
            else{
                throw new IOException("ioE");
            }

        }
        catch(Exception e){
            return 2;
        }
        finally{
            return 3;
        }
    }


    public static void main(String args[]){
        int a = 0;
        try{
         a = devuelveEntero(1);
        }catch(IOException ioE){}
        System.out.println(a);
    }
}
```

¿Qué salida se obtiene por pantalla tras su ejecución?

Solución: La llamada al método `devuelveEntero()` se invoca utilizando un número impar como argumento. En el método `main()` habrá que capturar la excepción aunque no se produzca. Nuevamente la inclusión del bloque `finally{}` hace que su bloque se ejecute. De esta forma la variable `a` recibirá el valor 3 que será el que se muestre por pantalla.

Ejercicio 4.38:

¿Qué salida por pantalla mostrará la ejecución del siguiente programa?

```
public class EjemploExcepciones3 {

    public static int devuelveEntero(int num) throws IOException{
        try{
            if(num%2==0){
```

```
            throw new Exception("Lanzando excepción");
          }
          else{
              throw new IOException("Desde try");
          }

      }
      catch(Exception e){
          return 2;
      }
      finally{
         throw new IOException("Desde finally");
      }
  }

  public static void main(String args[]){
      int a = 0;
      try{
       a = devuelveEntero(1);
      }catch(IOException ioE){
          System.out.println(ioE.getMessage());
      }
      System.out.println(a);
  }
}
```

Solución:
El mensaje que se mostrará por pantalla en el manejador de la excepción del método `main()` será `"Desde finally..."` y se mostrará el valor 0 para la variable `a`. La presencia del bloque `finally{}` dará lugar a que se genere su excepción. La excepción lanzada desde el bloque `try{}` se perderá.

Ejercicio 4.39:

Escriba un programa que genere números aleatorios hasta obtener primero un array de 5 números pares y, después, un array de 5 números impares.

Planteamiento: El método declara, en primer lugar, los arrays donde se almacenarán los números pares e impares. Se utilizará la variable referencia `arr` para apuntar, primero, al array correspondiente a los números pares y, una vez obtenido éste, al de los números impares. La variable `resto` almacenará el resto que debe obtenerse para clasificar el número como par o impar. El ejercicio se ha resuelto utilizando dos bloques `try{}catch{}` anidados. El bloque más externo controla la salida definitiva del bucle infinito que obtiene números aleatorios y verifica si son pares o impares. El bloque más interno se utiliza para capturar la excepción que se lanzará cuando se obtenga el array de números pares. El manejador para esta excepción cambia la referencia `arr` al array de impares, pone a 0 el índice del array (variable i) y pone a 1 el resto que debe obtenerse en la división entre 2 del número aleatorio (ahora se buscan valores impares). Cuando el array de números impares se haya completado se lanzará de nuevo un objeto `Exception`, pero ahora el manejador actúa lanzando otra excepción que se capturará en el bloque más externo y pondrá fin a la ejecución del bucle.

Solución:

```
public static void main(String args[]){
        int[] impares= new int[5];
        int[] pares= new int[5];
```

```
        int[] arr=pares;
        int resto=0;
        int i=0;
        try{
           while(true){
              try{
                 int numero=(int)(Math.random()*100);
                 if(numero % 2 == resto ){
                       array[i++]=numero;
                 }
                 if(i==5){
                       throw new Exception(""+
                                      (resto == 0 ? "pares" : "impares") +
                                      " generados");
                 }
              }catch(Exception e){
                 System.out.println(e.toString());
                 if(resto==0){
                        resto=1;
                        array=impares;
                        i=0;
                 }else{
                        throw new Exception("Fin");
                 }
                }
              }
          }catch(Exception e){
                 System.out.println(e.getMessage());
          }
        System.out.println("Valores pares: " + Arrays.toString(pares));
        System.out.println("Valores impares: " + Arrays.toString(impares));
   }
```

Excepción que notifica la obtención del array de valores pares e impares.

La primera vez que se captura una excepción se realizan ajustes para obtener el array de impares. La segunda vez se lanza otra excepción que será capturada en el bloque más externo.

Aviso: Evidentemente hay mejores formas de hacerlo. Sólo permite ilustrar cómo el control de las excepciones también permite controlar el flujo de ejecución del programa.

Ejercicio 4.40:

Escriba una clase de nombre ClaseExcep1 que incluya un método denominado dividirEntreArray. Este método recibirá por parámetro un número entero y un array de elementos del mismo tipo. El método mostrará por pantalla el resultado de la división entre el número recibido y cada uno de los elementos del array. A continuación, se escribirá un programa que invoque al método con el número 2 y un array con los elementos -2, -1, 0, 1 y 2.

Planteamiento: Se escribe la clase con el método solicitado. Este método utiliza un bucle para acceder a todas las posiciones del array recibido. En cada iteración del bucle se mostrará por pantalla el resultado de la división entera entre el número recibido y el ocupado por la correspondiente posición del array. En el método `main()` de la clase `PruebaExcep1` se crea un objeto de la clase anterior y se invoca sobre él al método `dividirEntreArray` con los argumentos especificados en el enunciado.

Solución:

```
class ClaseExcep1{
    public void dividirEntreArray(int num, int[] arr){
```

```
        for(int i: arr){
            System.out.println("La división entera entre " + num +
                               " y: " + i +
                               " es: " + num/i);
        }
    }
}
class PruebaExcep1{
    public static void main(String args[]){
        int[] arr = {-2, -1, 0, 1 ,2};
        ClaseExcep1 pe = new ClaseExcep1();
        pe.dividirEntreArray(2, arr);
    }
}
```

La invocación de este método causará una excepción de tipo ArithmeticException.

Comentario: Las dos clases anteriores compilan sin generar ningún error de compilación. Cuando se ejecuta el método `main()` de la clase `PruebaExcep1`, se obtendrá por pantalla el siguiente resultado:

```
La división entera entre 2 y: -2 es: -1
La división entera entre 2 y: -1 es: -2
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at ejexceptions.PruebaExcep1.dividirEntreArray(EjExceptionsBasicos.java:19)
    at ejexceptions.EjExceptionsBasicos.main(EjExceptionsBasicos.java:98)
Java Result: 1
```

El intento de dividir el valor 2 suministrado como argumento entre el contenido de la tercera posición del array (valor 0) da lugar al lanzamiento de la excepción `ArithmeticException` que causa la terminación del programa. Esta excepción deriva de una clase de excepciones especial denominada `RuntimeException` que representa algunos de los errores de programación más comunes. Dado que se considera que cualquier método o programa puede ocasionar el lanzamiento de estas excepciones, no se obliga a definir un manejador específico para su tratamiento. Piense que ello equivaldría a asumir que se está cometiendo un error y a incluir las sentencias necesarias para tratarlo. Más sencillo sería no cometer directamente el error. No obstante, en ciertas ocasiones, sí puede ser de utilidad definir un manejador para estas excepciones para evitar la terminación abrupta del programa. Si no se define un manejador específico para la excepción, se utilizará el predeterminado por Java que consiste en mostrar el tipo de excepción y la secuencia de llamadas a métodos que han provocado su lanzamiento.

Ejercicio 4.41:

Reescriba el método anterior para capturar la excepción derivada del intento de división entre 0, de forma que no se interrumpa la ejecución del programa y se continúe dividiendo entre los otros elementos del array.

Planteamiento: Se modifica el método `dividirEntreArray()` para que capture la excepción `ArithmeticException` añadiendo el correspondiente bloque `try{}catch{}`.

Solución:

```
class PruebaExcep1{
    public void dividirEntreArray(int num, int[] arr){
        for(int i: arr){
            try{
            System.out.println("La división entera entre " + num +
                               " y: " + i +
```

```
                        " es: " + num/i);
        }catch(ArithmeticException aE){
            System.out.println("No se puede dividir entre 0");
        }
      }
    }
}
```

Captura de la excepción de tipo ArithmeticException.

Comentario: Frente a este tipo de excepciones, resulta, quizá, más conveniente incluir las sentencias de código necesarias para que la excepción no llegue a producirse. En este caso bastará una sencilla comprobación de que el divisor de la excepción nunca puede tratarse de un 0.

```
class PruebaExcep1{
    public void dividirEntreArray(int num, int[] arr){
      for(int i: arr){
        if(i != 0){
        System.out.println("La división entera entre " + num +
                            " y: " + i +
                            " es: " + num/i);
        }else{
            System.out.println("No se puede dividir entre 0");
        }
      }
    }
}
```

Ejercicio 4.42:

Escriba un método, de nombre mostrarCadenasArray, que reciba por parámetro un array con cadenas de caracteres. El método mostrará por pantalla el primer carácter de cada una de las cadenas contenidas en el array.

Planteamiento: El programa utiliza un bucle para recorrer todas las cadenas contenidas en el array. Para obtener el primer carácter de cada cadena utiliza el método `charAt()` de la clase `String`, proporcionando como argumento el valor de índice 0.

Si alguna de las posiciones del array contuviera una referencia no inicializada, es decir, con valor `null`, la invocación del método sobre tal referencia provocaría una excepción de la clase `RuntimeException` de nombre `NullPointerException`. Para prevenir su lanzamiento se comprueba previamente que la referencia no es nula.

Parámetros: Array de cadenas cuyo primer carácter se mostrará por pantalla.

Valor de retorno: El método no devuelve ningún valor de retorno, por lo que se utiliza la cláusula `void`.

Solución:

```
public void mostrarPrimerCarCadenas(String[] arrCad){
  for(String cad: arrCad){
      if (cad != null){
          System.out.println(cad.charAt(0));
      }
  }
}
```

Se comprueba que la cadena no es nula para evitar que se lance una excepción *NullPointerException*.

Ejercicio 4.43:

Escriba un método, de nombre abrirFlujoLecturaArchivo, que reciba el nombre de un archivo y trate de abrir un flujo de lectura sobre él devolviendo el objeto que lo represente. El método no capturará ninguna excepción.

Planteamiento: El método abre un flujo de lectura sobre el nombre del archivo recibido por parámetro. El flujo quedará representado por un objeto de la clase `FileReader` que será devuelto a continuación. La excepción `FileNotFoundException` que puede lanzarse en el momento de la lectura no va a ser capturada por el método, por esta razón su cabecera debe incluir la cláusula `throws` para indicar que la invocación del método puede originar esta excepción y, por tanto, será recibido dentro del método invocante.

Parámetros: Cadena de caracteres con el nombre del archivo sobre el que se abrirá el flujo de lectura.

Valor de retorno: Objeto de la clase `FileReader` que representa el flujo de lectura abierto sobre al archivo recibido.

Solución:

```
public FileReader abrirFlujoLecturaArchivo(String nombreArchivo)
                              throws FileNotFoundException{ ←
    FileReader lectura = new FileReader(nombreArchivo);
    return lectura;
}
```

El método delega la excepción *FileNofFoundException*, si ésta se produce.

Ejercicio 4.44:

Añada el método anterior a una clase, de nombre EjemploDelegacion, y escriba un nuevo método de nombre leerPrimerCaracterArchivo. El método capturará la excepción IOException que pueda originar la lectura del carácter, haciendo que el manejador devuelva el carácter 'A' en tal caso. Sin embargo, el método delegará la excepción FileNotFoundException originada si el archivo recibido no se encuentra disponible para la lectura.

Planteamiento: El método invoca al método `abrirFlujoLecturaArchivo()` con el nombre del archivo recibido. Una vez obtenido el flujo, trata de leer el primer carácter de él y lo devuelve. El método captura la excepción `FileNotFoundException` que puede derivarse de la apertura del flujo y la relanza dentro del correspondiente manejador. También captura la excepción `IOException` para devolver en su manejador el carácter '\0'.

Parámetros: Nombre del archivo del que se va a leer su primer carácter.

Valor de retorno: Primer carácter leído del archivo recibido por parámetro.

Solución:

```
public class EjemploDelegación {

    public FileReader abrirFlujoLecturaArchivo(String nombreArchivo)
                                              throws FileNotFoundException{
        FileReader lectura = new FileReader(nombreArchivo);
        return lectura;
    }

    public char leerPrimerCaracterArchivo(String archivo) throws FileNotFoundException{
        try{
            FileReader flujo = abrirFlujoLecturaArchivo(archivo);
```

```
                char car = (char)flujo.read();
                return car;
            }
            catch(FileNotFoundException fnfE){
                throw new FileNotFoundException(fnfE.getMessage());
            }
            catch(IOException ioE){
                return '\0';
            }
        }
    }
```

Relanzamiento de la excepción *FileNotFoundException.*

Comentario: La invocación al método `abrirFlujoLecturaArchivo()` puede generar una excepción de tipo `FileNotFoundException`, como el método no debe capturarla se incluye un bloque simplemente para capturarla y volver a lanzarla (relanzamiento).

Si no se introdujera este bloque la excepción sería capturada por el bloque que captura la excepción `IOException`, ya que deriva de ella, y se llevaría a cabo la misma acción que si se produce un error de lectura.

Ejercicio 4.45:

Añada un método a la clase anterior, de nombre mostarPrimerCaracterArchivos, que reciba por parámetro un array de nombres y muestre por pantalla el primer carácter de cada uno de ellos. El método capturará todas las excepciones que puedan producirse.

Planteamiento: El método recorre el array con los nombres de los archivos e invoca al método `leerPrimerCaracterArchivo()` sobre cada uno de ellos. Como este método puede lanzar una excepción de tipo `FileNotFoundException` es necesario incluir un bloque `try{}catch{}` para su captura. El método no podrá recibir ninguna excepción de la clase `IOException` ya que éstas son capturadas por el método `leerPrimerCaracterArchivo()`.

Parámetros: Array con los nombres de los archivos de los que se va a mostrar su primer carácter.

Valor de retorno: El método no devuelve ningún valor de retorno, por lo que se utiliza la cláusula `void`.

Solución:

```
public class EjemploDelegación {

    public void mostrarPrimerCaracterArchivos(String[] archivos){
        for(String archivo: archivos){
            try{
                char car= leerPrimerCaracterArchivo(archivo);
                System.out.println("Primer carácter del archivo: "+
                                   archivo + ": " + car);
            }
            catch(FileNotFoundException fnfE){
                System.out.println("No se pudo abrir el archivo: " +
                                   archivo + " para lectura");
            }
        }
    }
}
```

Captura de la excepción *FileNotFoundException.*

Comentario: Fíjese que la excepción `FileNotFoundException` es delegada por el método `abrirFlujoLecturaArchivo()`. El método `leerPrimerCaracterArchivo()` la relanza y finalmente este método la captura.

Ejercicio 4.46:

Escriba una clase, de nombre PruebaDelegación, en cuyo método main() se instanciará un objeto de la clase EjemploDelegación sobre el que se invocará al método mostrarPrimerCaracterArchivos pare verificar su funcionamiento.

Planteamiento: Se crea un array con los nombres de dos archivos. Se procede a instanciar un objeto de la clase `EjemploDelegacion` sobre el que se invoca el método `mostrarPrimerCaracterArchivos()`.

Solución:

```
public class PruebaDelegación {
public static void main(String args[]){
        String[] archivos = {"C:/misArchivos/archivo1.txt",
                             "C:/misArchivos/archivo2.txt"};
        EjDelegacion d = new EjDelegacion();
        d.mostrarPrimerCaracterFicheros(archivos);
    }
}
```

Comentario: El método `main()` no incluye ningún bloque `try{}catch{}` ya que todas las excepciones que se hayan lanzado se habrán capturado en los métodos invocados.

Ejercicio 4.47:

Escriba un método, de nombre leerPrimeraLineaArchivo, que reciba un nombre de archivo y devuelva su primera línea. El método no capturará ninguna excepción.

Planteamiento: La cabecera del método incluye la cláusula `throws` para indicar que delega cualquier excepción de tipo `IOException`. El método abre un flujo de lectura utilizando la clase `BufferedReader` para poder leer una línea completa. El método lee la primera línea del archivo y la devuelve después de cerrar el flujo de lectura.

Parámetros: Nombre del archivo del que se leerá su primera línea.

Valor de retorno: Cadena de caracteres correspondiente a la primera línea del archivo.

Solución:

```
public String leerPrimeraLineaArchivo(String archivo) throws IOException{
    BufferedReader lectura = new BufferedReader(
                                 new FileReader(archivo));
    String pLinea = lectura.readLine(); <----- Lectura de la primera línea del archivo.
    lectura.close();
    return pLinea;
}
```

Comentario: El método no incluye ningún bloque `try{}catch{}` ya que todas las excepciones producidas se delegan.

Ejercicio 4.48:

Añada el método anterior a la clase EjemploDelegacion2 y escriba una nueva clase de nombre PruebaDelegacion2 en cuyo método main() se utilice el método anterior.

Planteamiento: El método `main()` de la clase `PruebaDelegacion2` instancia un objeto de la clase `EjemploDelegacion2` e invoca sobre él al método `leerPrimeraLineaArchivo()`.

Solución:

```
public class EjemploDelegacion2 {
    public String leerPrimeraLineaArchivo(String archivo) throws IOException{
        BufferedReader lectura = new BufferedReader(
                                        new FileReader(archivo));
        String pLinea = lectura.readLine();
        lectura.close();
        return pLinea;
    }
}
public class PruebaDelegacion2 {
    public static void main(String args[]){
        EjemploDelegacion2 d = new EjemploDelegacion2();
        try{
            System.out.println(d.leerPrimeraLineaArchivo("C:/misArchivos/archivo1.txt"));
        }
        catch(IOException ioE){
            System.out.println("Error de lectura: " + ioE.getMessage());
        }
    }
}
```

Bloque *try{}catch{}* para la captura de las excepciones *IOException*.

Comentario: Dado que el método `leerPrimeraLineaArchivo()` delega las excepciones `IOException`, el método `main()` debe incluir un bloque `try{}catch{}` para capturarlas.

Ejercicio 4.49:

Escriba un método, de nombre enviarMensaje, que reciba por parámetro una cadena de caracteres correspondiente a una dirección de correo electrónico. El método comprobará que la dirección recibida es correcta elevando la excepción DirCorreoIncorrectaExcepcion en caso contrario. La comprobación consistirá en verificar que la dirección contiene el carácter arroba (@), algún carácter después de él antes del carácter punto (.) y algún carácter después de éste

Planteamiento: En primer lugar se crea una nueva clase de nombre `DirCorreoIncorrectaExcepcion` que corresponderá a la situación en que una dirección de correo no sea correcta. Se hace que esta clase herede de la clase `Exception` (los mecanismos de herencia se estudiarán más adelante). Su constructor simplemente recibe un mensaje con el que invoca al constructor de la superclase para asociar el mensaje recibido a la excepción. El método comprueba la validez de la dirección utilizando los métodos `lastIndexOf()` que proporciona el índice de la última ocurrencia de un carácter en la cadena. Si la dirección recibida no cumple las especificaciones del enunciado se eleva la excepción `DirCorreoIncorrectaExcepcion`. El desarrollo del resto del método no resulta relevante.

Solución:

```
class DirCorreoIncorrectaExcepcion  extends Exception{
    public DirCorreoIncorrectaExcepcion(String msj){
        super (msj);
    }
}
```

```
    public void enviarMensaje(String dir) throws DirCorreoIncorrectaExcepcion{
        if(!(dir.lastIndexOf('@') != -1 &&
            dir.lastIndexOf('.') > dir.lastIndexOf('@')+1 &&
            dir.lastIndexOf('.') < dir.length() )){
                throw new DirCorreoIncorrectaExcepcion("Direccion incorrecta");
        }

        System.out.println("Direccion: "+ dir);
    }
```

Ejercicio 4.50:

Escriba una clase, de nombre Corredor, teniendo en cuenta las siguientes especificaciones:

a) La clase tendrá un atributo entero de nombre energía.
b) La clase tendrá un método constructor que reciba por parámetro una cantidad de energía que asignará al atributo.
c) La clase tendrá un método, de nombre recargarEnergía, que recibirá por parámetro una cantidad de energía que será sumada al atributo energía.
d) La clase tendrá un método, de nombre correr, que mostrará por pantalla un mensaje y decrementará la energía en 10 unidades. Antes de proceder al decremento, el método comprobará que la energía del corredor es igual o superior a 10 unidades. Si no es así, el método lanzará una excepción de tipo AgotadoExceptíon.

Planteamiento: En primer lugar se crea una nueva clase de nombre AgotadoExcepcion que corresponderá a la situación en que el corredor se quede sin energía. Se hace que esta clase herede de la clase Exception (los mecanismos de herencia se estudiarán más adelante). Su constructor simplemente recibe un mensaje con el que invoca al constructor de la superclase para asociar el mensaje recibido a la excepción.

En la clase Corredor se declara su atributo energía como entero. Su constructor recibe una cantidad de energía que asigna al atributo. El método recargarEnergía() recibe una cantidad de energía que suma al atributo. Por último, el método correr comprueba, en primer lugar, que el corredor dispone de la suficiente energía para correr. En caso contrario, se lanzará la excepción AgotadoExcepcion con el mensaje "Estoy agotado...". Si el corredor puede correr se mostrará el mensaje "Estoy corriendo..." y su energía se decrementará en 10 unidades.

Solución:

```
class AgotadoExcepcion extends Exception{
    public AgotadoExcepcion(String msj){
        super (msj);
    }
}
public class Corredor {
    int energía;

    public Corredor(int energía) {
        this.energía = energía;
    }

    public void recargarEnergía(int energía){
        this.energía += energía;
    }
   public void correr() throws AgotadoExcepcion { 
        if(energía < 10){
```

El método delega la excepción *AgotadoExcepcion*.

```
            throw new AgotadoExcepcion("Estoy agotado....");
        }
        System.out.println("Estoy corriendo.....");
        Energía -= 10;
    }
}
```

Ejercicio 4.51:

Escriba una clase, de nombre Entrenamiento, en cuyo método main() se creará un objeto Corredor con una energía de 50 unidades. Se hará que el corredor corra hasta que se agote 3 veces. La primera vez que se agote, su energía se recargará con 30 unidades. La segunda vez que se agote su energía se recargará con 10 unidades. Cuando el corredor se agote por tercera vez se dará el entrenamiento por concluido.

Planteamiento: En el método `main()` se crea un objeto corredor con energía de 50 unidades, se declara una bandera booleana de nombre `seguir` para determinar cuándo finaliza el entrenamiento. Dentro de un bucle gobernado por la bandera anterior se pone a correr al corredor. Se invocará al método correr dentro de un bucle `try{}catch{}` para capturar la excepción `AgotadoExcepcion`. Dentro del bloque `catch{}` se irán contabilizando las veces que el corredor se agota para determinar si se recarga de nuevo su energía o se da por finalizado el entrenamiento.

Solución:

```
public class Entrenamiento{
    public static void main(String args[]){
        Corredor c = new Corredor(50);
        boolean seguir = true;
        int numVecesAgotado=0;
        do{
            try{
                c.correr();
            }
            catch(AgotadoExcepcion aE){ <----
                numVecesAgotado++;
                switch(numVecesAgotado){
                    case 1: c.recargarEnergía(30);
                            System.out.println(aE.getMessage());
                            break;
                    case 2: c.recargarEnergía(10);
                            System.out.println(aE.getMessage());
                            break;
                    case 3: seguir=false;
                            System.out.println("Entrenamiento Terminado");
                }
            }
        }while(seguir);
    }
}
```

El bloque `catch{}` controla las veces que el corredor se agota.

Aserciones

Ejercicio 4.52:

Dada la siguiente clase que se ha escrito para un programa, indique para qué sirven las aserciones incluidas en el método imprime().

```
public class Persona{

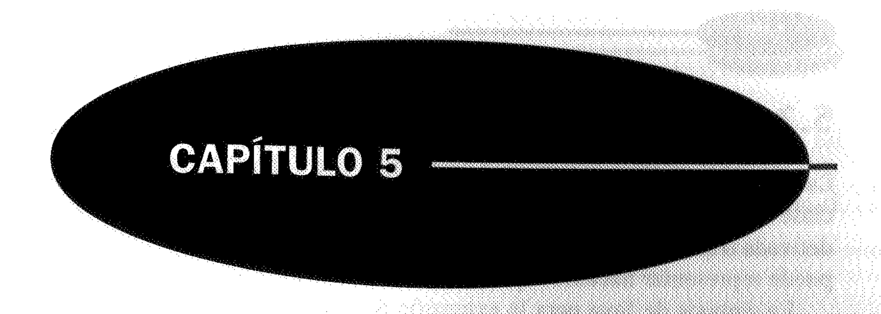
    private String nombre;
    private String apellidos;
    private int añoDeNacimiento;

    public Persona(String nombre, String apellidos, int añoDeNacimiento){
        setNombre(nombre);
        setApellidos(apellidos);
        setAñoDeNacimiento(añoDeNacimiento);
    }

    // Imprime por pantalla los datos personales
    public void imprime() {
        assert nombre != null && apellidos != null : "El nombre y los apellidos no pueden ser nulos.";
        assert nombre.length > 1 && apellidos.length > 1 :
              "El nombre y los apellidos deben tener al menos dos caracteres.";
        assert añoDeNacimiento > 1900 : "El año de nacimiento es anterior a 1900.";
        System.out.print("Datos personales: " + nombre +
                         " " + apellidos +
                         " (" + añoDeNacimiento +")");
    }
}
```

Solución: Las aserciones se utilizan para asegurar los valores del estado del objeto o de los invariantes de un algoritmo. En el caso del ejemplo, las aserciones permiten asegurar que los valores de nombre, apellidos y año de nacimiento tienen unos valores correctos, es decir, permiten asegurar que el estado del objeto es apropiado y no se ha modificado de forma no prevista.

Nota: Puede ver otro uso de las aserciones en el Ejemplo 5.10 del Capítulo 5.



Extensión de clases

5.1 COMPOSICIÓN

La composición es el agrupamiento de uno o varios objetos y valores, como atributos, que conforman el valor de los distintos objetos de una clase. Normalmente los atributos contenidos se declaran con acceso privado (`private`) y se inicializan en el constructor de la nueva clase.

Un ejemplo puede ser la definición de una clase Claustro, donde un claustro está formado por un Catedrático, un Profesor Titular, un Alumno, un Representante de la Administración y el Rector. En la siguiente definición, se suponen declaradas las clases necesarias para dichos objetos:

```
class Claustro {
    private Profesor catedrático, titular, rector;
    private Alumno alumno;
    private Personal repAdmon;

    // Construcción de los objetos contenidos
    // en el constructor de la clase contenedora
    Claustro(Profesor rector, Profesor catedrático, Profesor titular,
            Alumno alumno, Personal repAdmon) {
        this.rector      = rector;
        this.catedrático = catedrático;
        this.titular     = titular;
        this.alumno      = alumno;
        this.repAdmon    = repAdmon;
    }
}
```

La composición crea una relación *tiene*. Así, por ejemplo, un Automóvil, tiene un número de bastidor, una matrícula, una fecha de compra, un dueño, etc. Todos esos elementos se representan utilizando el mecanismo de composición sobre la clase Automóvil, de la misma forma que se ha hecho en el ejemplo anterior con la clase Claustro.

⇨ Sobre la composición de clases consulte los Ejercicios 5.1 al 5.6.

5.2 HERENCIA

La herencia establece una relación *es-un* entre clases. La herencia introduce la capacidad de extender clases, donde la clase original se denomina clase padre (o madre), clase base o superclase, y la nueva clase se denomina clase hija, derivada o subclase. Así, una clase derivada *es-una* clase base. Por ejemplo, Alumno es-una Persona. Esta relación se puede representar haciendo que la clase Alumno extienda (herede) de la clase Persona.

La sintaxis de Java para la extensión de clases (herencia) es la siguiente:

```
class ClaseDerivada extends ClaseBase { ... }
```

En el caso del Alumno se escribiría:

```
class Alumno extends Persona { ... }
```

La nueva clase Alumno, por ser una extensión de Persona tiene ya todos los atributos y métodos de Persona y puede acceder a todos ellos como si estuviesen declarados en Alumno, excepto a los que se hayan definido como privados en Persona. Adicionalmente, en la clase Alumno se pueden declarar todos los atributos y métodos necesarios que tenga un Alumno y que no estén definidos en la clase Persona. Para aclarar el ejemplo con respecto a los atributos, las definiciones de Persona y Alumno podrían ser las siguientes:

```
class Persona {
    private String nombre;
    private String apellidos;
    private int añoDeNacimiento;

    // Imprime por pantalla los datos personales
    // con el formato:
    // Datos personales: nombre apellido (añoNaci.)
    public void imprime() {
        System.out.print("Datos personales: " + nombre +
                         " " + apellidos +
                         " (" + añoDeNacimiento +")");
    }
    // resto de métodos de la clase
}
```

La clase Alumno hereda de esta clase Persona por lo que, al heredar sus atributos y métodos, un alumno ya tiene nombre, apellidos y año de nacimiento y no hay que declararlos en la clase Alumno. Los atributos y métodos que se añadan a la clase Alumno, se suman a todos lo que ya posee como Persona. La declaración de la clase Alumno podría ser:

```
class Alumno extends Persona {
    protected String  grupo;
    protected Horario horario;

    public void ponGrupo(String grupo, Horario horario) throws Exception {
        if (grupo == null || grupo.length() == 0)
            throw new Exception("Grupo no válido.");
        this.grupo   = grupo;
        this.horario = horario;
    }
}
```

El uso de ambas clases sigue las normas usuales, pero hay que tener en cuenta que en la clase Alumno hay un comportamiento que no está explícitamente escrito. Así, se puede crear un alumno e invocar el método imprime() de la clase base. Al ejecutarse el método imprime(), como ese método no se ha escrito en la clase Alumno, se ejecuta el escrito en la clase Persona. Por tanto, es correcto escribir:

```
Alumno unAlumno = new Alumno();
// se inicializan sus datos
unAlumno.imprime();
```

⇨ Sobre los principios de herencia y su uso consulte los Ejercicios 5.7 al 5.14.

5.3 COMPATIBILIDAD DE TIPOS

La herencia establece una relación *es-un*, es decir, un Alumno *es-una* Persona. En este sentido cualquier objeto Alumno también se puede considerar Persona. Esto quiere decir que una variable de la clase Persona puede contener una referencia a un objeto Persona o a un objeto de cualquier clase derivada de Persona. A esta compatibilidad se le denomina compatibilidad ascendente y es automática.

```
Persona p1;
Alumno Alumno1 = new Alumno("Alicia", "Moreno", 1990);
p1 = Alumno1;  // conversión ascendente
Persona p2 = new Alumno("Blanca", "Moreno", 1994);
```

También se puede realizar la asignación de una variable de la clase Persona a una variable de la clase Alumno, siempre que la variable de la clase Persona guarde una referencia a un objeto Alumno o derivado de Alumno. Esta conversión hay que indicarla explícitamente. A esta compatibilidad se le denomina compatibilidad descendente y si no es correcta se lanza la excepción ClassCastException.

```
Alumno otroAlumno;
otroAlumno = (Alumno) p1;  // p1 era un objeto Alumno creado antes
```

Para comprobar si se puede realizar una conversión descendente se utiliza el operador instanceof. Se evalúa a true si el objeto es instancia de la clase o false en caso contrario:

```
if (p1 instanceof Alumno)
    otroAlumno = (Alumno) p1; // no lanzará la excepción, ya se ha comprobado que se puede
```

⇨ Sobre la compatibilidad ascendente y descendente de clases consulte el Ejercicio 5.16.

5.4 ÁMBITOS Y VISIBILIDAD

El ámbito de las clases derivadas funciona de igual forma que para cualquier clase. Heredan el ámbito de la clase base, por lo que se pueden usar los atributos y los métodos que tuviera la clase base excepto los calificados como privados (private).

En cuanto al modificador de acceso protegido (protected) restringe la visibilidad de forma que sólo la propia clase que los define, sus derivadas y las clases del mismo paquete pueden usarlos. Por tanto, los miembros protegidos son como públicos para las clases derivadas y las clases del mismo paquete y como privados para el resto de las clases.

⇨ Sobre la modificación de la visibilidad en clases derivadas consulte el Ejercicio 5.15.

5.5 SOBREESCRITURA

En una clase derivada se puede volver a escribir uno de los métodos que ya tuviese la clase base de la que hereda. A este mecanismo se denomina sobreescritura, reescritura o redefinición dependiendo de los autores. Así mismo, también se puede sobrescribir los atributos de la clase, para ello una clase derivada puede definir:

- Un atributo con el mismo nombre que uno de la clase base.
- Un método con la misma signatura (nombre y parámetros de la llamada) que uno de la clase base.

Aunque los elementos de clase (estáticos) se pueden sobreescribir, puesto que no forman parte de las instancias de la clase al utilizarlos, se accede de acuerdo a la declaración del objeto con que se llaman sin que exista ligadura dinámica con ellos, ya que no necesitan objetos para utilizarlos.

En la sobreescritura se puede ampliar el nivel de acceso, haciéndolo más público, pero no restringirlo haciéndolo más privado. En la Tabla 5.1 se muestran los cambios posibles en el acceso.

Tabla 5.1. Cambios posibles en el acceso.

Derecho en clase base	Puede ser en clase derivada a
privado	privado protegido de paquete público
de paquete	de paquete protegido público
protegido	protegido público
público	público

Es muy común que al reescribir en la clase derivada métodos de la clase base, sea necesario invocar los métodos originales de la clase base. Para ello, se dispone de la referencia `super`, que permite acceder a los métodos y atributos de la clase base. Ello permite utilizar sentencias como:

```
super.imprime(); //llama al método imprime de la clase base
```

Para impedir que un atributo o método se pueda reescribir en una clase derivada, en la clase base se le antepone el modificador `final`. Este modificador `final` permite, dependiendo del elemento al que se aplique:

- Para una variable: se impide que se cambie su contenido (constante).
- Para un método: se impide que se sobrecargue.
- Para un parámetro de un método: impide que el método cambie su valor.
- Para una clase: impide que se herede de ella.

⇨ Sobre la sobreescritura de métodos consulte los Ejercicios 5.15 y 5.18.

5.6 CONSTRUCTORES

Crear objetos de una clase derivada sigue un proceso ordenado que debe conocer. Al crear un objeto de una clase derivada, primero se crea la parte correspondiente a la clase base y, después, se construye la parte de la clase derivada.

Si la clase base es, a su vez, derivada de otra, se aplica el mismo proceso, hasta llegar al constructor de Object. El proceso se complica cuando se añaden constructores a las clases. Cuando una clase contiene un constructor, primero se construye la clase base, luego la clase derivada y entonces se ejecuta el código del constructor. La clase base se construye con el constructor por defecto si existe. Si en la construcción de la clase derivada se desea utilizar uno determinado de la clase base, se indica con la sentencia súper(parámetros).

Por ejemplo en la clase Persona y la clase Alumno:

```
class Persona {
    private String nombre;
    private String apellidos;
    private int añoDeNacimiento;

    Persona(String nombre,
            String apellidos,
            int añoDeNacimiento) {
        this.nombre = nombre;
        this.apellidos = apellidos;
        this.añoDeNacimiento = añoDeNacimiento;
    }
}

class Alumno extends Persona {
    private String  grupo;
    private Horario horario;

    Alumno(String nombre,
           String apellidos,
           int añoDeNacimiento) {
        super(nombre, apellidos, añoDeNacimiento);
    }
}
```

Un objeto de la clase Alumno se construye con un constructor específico de la clase Persona. Si hubiera dos o más constructores en la clase Persona, se podría invocar a cualquiera de ellos con super(), pero sólo a uno. Según el esquema indicado la sentencia super() tiene que ser obligatoriamente la primera llamada de un constructor.

⇨ Sobre el uso de super y la invocación de otros constructores, consulte los Ejercicios 5.7, 5.10 y 5.13.

5.7 POLIMORFISMO

De acuerdo con la compatibilidad ascendente, se puede asignar a una variable de una clase una referencia de cualquier objeto de la misma clase o de cualquier clase derivada de ella. Por ejemplo, suponiendo que no se añaden constructores a ninguna de las clases en el ejemplo conductor para simplificar, es posible escribir:

```
Persona p = new Alumno();
Alumno  a = new Alumno();
```

Se va a suponer que la clase Persona dispone de un método imprime() que se ha sobreescrito en la clase Alumno. Entonces se puede escribir:

```
p.imprime();
a.imprime();
```

Ambas invocaciones del método `imprime()` ejecutarán el código de la clase `Alumno`, pues las referencias `p` y `a` contienen instancias de la clase `Alumno`, aunque `p` esté declarada como `Persona`. Ahora bien, esto sólo es posible porque el método `imprime()` existe también en la clase `Persona`. A este mecanismo de asociación del código que se ejecuta se denomina ligadura dinámica.

⇨ Sobre el polimorfismo y la ligadura dinámica consulte los Ejercicios 5.12, 5.14 y 5.16.

5.8 HERENCIA FORZADA Y CLASES ABSTRACTAS

Se puede obligar a que para usar una clase haya que hacerlo escribiendo una nueva clase que herede de ella. Para ello se utiliza en modificador `abstract` antes de la definición de la clase:

```
abstract class NombreDeClase { ... }
```

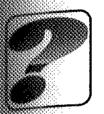
Una clase declarada abstracta no se puede instanciar, aunque pueden existir referencias con objetos de clases derivadas por compatibilidad ascendente, lo que quedaría de la siguiente forma:

```
NombreDeClase objeto = new NuevaClase();
```

Siendo `NuevaClase` una clase derivada de `NombreDeClase`.

Si se utiliza el modificador `abstract` delante de la cabecera de un método, éste es abstracto, con lo que la clase también lo será. Con ello se consigue tener realizaciones parciales de tipos de objetos cuya codificación no existe. Una clase abstracta es, por tanto, una clase especificada pero no totalmente implementada. Lo que está obligando a que todas las clases derivadas ofrezcan una parte común en su interfaz.

⇨ Sobre la herencia forzada y los métodos abstractos consulte los Ejercicios 5.20 y 5.21.



Problemas resueltos

Composición

Ejercicio 5.1:

Realice una clase, de nombre Examen, para guardar información sobre los exámenes de un centro educativo. La información que se guarda de un examen es: el nombre de la asignatura, el aula, la fecha y la hora. Para guardar la fecha y la hora hay que realizar dos clases, Fecha y Hora.

La clase Fecha guarda día, mes y año. Todos los valores se reciben en el constructor por parámetro. Además, esta clase debe tener un método que devuelva cada uno de los atributos y un método toString() que devuelva la información de la fecha en forma de String.

La clase Hora guarda hora y minuto. También recibe los valores para los atributos por parámetro en el constructor, tiene métodos que devuelven cada uno de los atributos y un método toString().

Planteamiento: La clase `Fecha` tiene tres atributos de tipo entero, cuyos valores se reciben en el constructor por parámetro. El método `toString()` devuelve los valores de los tres atributos separados por una barra (`"/"`). La clase `Hora` tiene dos atributos de tipo entero, cuyos valores se reciben en el constructor por parámetro. El método `toString()` devuelve los dos atributos separados por dos puntos (`":"`).

La clase `Examen` realiza una composición de elementos de tipo `String`, `Fecha` y `Hora`. Recibe los valores de los atributos por parámetro en el constructor y además tiene un método para cambiar cada uno de los atributos y un método para devolver cada uno de los atributos. También tiene un método `toString()` que devuelve el valor de todos los atributos en forma de `String`.

Solución:

```
public class Fecha{

    private int día;
    private int mes;
    private int año;

    public Fecha(int día, int mes, int año){
        this.día = día;
        this.mes = mes;
        this.año = año;
    }

    public int getDía(){
        return día;
    }

    public int getMes(){
        return mes;
    }

    public int getAño(){
        return año;
    }

    public String toString(){
```

```
        return día + "/" + mes + "/" + año;
    }
}

public class Hora {

    private int hora;
    private int minuto;
    public Hora(int hora, int minuto) {
        if (hora < 0 || hora > 23 || minuto < 0 || minuto > 59){
            throw new IllegalArgumentException();
        }else{
            this.hora = hora;
            this.minuto = minuto;
        }
    }

    public int getHora(){
        return hora;
    }

    public int getMinuto(){
        return minuto;
    }

    public String toString(){
        return hora + ":" + (minuto < 10 ? "0" : "") +  minuto;
    }
}

public class Examen{
    private String asignatura;
    private String aula;
    private Fecha fecha;
    private Hora  hora;

    public Examen(String asignatura, String aula, Fecha fecha, Hora hora){
        this.asignatura = asignatura;
        this.aula   = aula;
        this.fecha = fecha;
        this.hora   = hora;
    }

    public void setAsignatura(String asignatura){
        this.asignatura = asignatura;
    }

    public void setAula(String aula){
        this.aula = aula;
    }
        public void setFecha(Fecha fecha){
```

Si la hora o el minuto no son correctos se lanza una excepción.

Los minutos tienen siempre dos dígitos. Si es menor que 10 se añade un 0.

Relación de composición con Fecha, Hora y String.

```
        this.fecha = fecha;
    }

    public void setHora(Hora hora){
        this.hora = hora;
    }

    public String getAsignatura(){
        return asignatura;
    }

    public String getAula(){
        return aula;
    }

    public Fecha getFecha(){
        return fecha;
    }

    public Hora getHora(){
        return hora;
    }

    public String toString(){
        return "Asignatura: " + asignatura +
               "\nAula: " + aula +
               "\nFecha: " + fecha.toString() +
               "\nHora: " + hora.toString();
    }
}
```

Comentario: En el constructor de la clase Hora se comprueba que los parámetros son correctos. Si no lo son se lanza una excepción.

En la clase Fecha se debería hacer lo mismo. En este ejercicio no se ha hecho porque alargaría el ejercicio sin aportar nada adicional de interés. Para comprobar que los datos son correctos se puede utilizar el método esFechaValida() del Ejercicio 4.12 del Capítulo 4 en el constructor, y si devuelve false lanzar una excepción. Como no hay método para modificar los atributos en estas dos clases, no sería necesario comprobar la validez en ningún otro método.

En el método toString() de la clase Hora se utiliza el operador ternario para añadir un 0 al principio de los minutos si es necesario. Para conseguir esto mismo, se puede usar la clase Formatter de la siguiente manera

```
public String toString(){
    Formatter f = new Formatter();
    return hora + ":" + f.format("%02d",minuto);   <-- Ancho mínimo 2, rellenar con ceros.
}
```

El método toString() pertenece a la clase Object, pero se recomienda que se sobreescriba en todas las clases. Cuando Java necesita un String y no encuentra un String si no un objeto, llama automáticamente al método toString() con ese objeto. Por ligadura dinámica se ejecutará el método toString() de la clase a la que pertenezca ese objeto.

Ejercicio 5.2:

Escriba una aplicación que cree un objeto de tipo Examen, lo muestre por pantalla, modifique su fecha y hora y lo vuelva a mostrar por pantalla.

Planteamiento: Para crear un objeto de tipo `Examen` hay que crear previamente los objetos de tipo `Hora` y `Fecha` que se pasan como argumentos. Posteriormente se crea el objeto de tipo `Examen`.

Para mostrar la información del examen por pantalla se utiliza el método `toString()` que devuelve la información, el resultado se pasa como argumento del método `System.out.println()` que muestra en la pantalla lo que recibe.

Para modificar la fecha y la hora se utilizan los métodos `setFecha()` y `setHora()` respectivamente, que reciben objetos de tipo `Fecha` y `Hora`. Se puede pasar la referencia que devuelve la creación del objeto directamente.

Solución:

```
public class Main {

    public static void main(String[] args) {
        Fecha fecha = new Fecha(1, 4, 2005);
        Hora hora   = new Hora(17, 30);
        Examen programación = new Examen("Programación en java", "A105", fecha, hora);
        System.out.println(programación.toString());
        programación.setFecha(new Fecha(1, 6, 2005));
        programación.setHora(new Hora(9, 0));
        System.out.println(programación.toString());
    }
}
```

Se pasa la referencia a `Fecha` que devuelve el constructor.

Ejercicio 5.3:

Escriba una aplicación para jugar al juego de los palillos. El juego de los palillos es un juego para dos personas que consta de un número determinado de palillos organizado por filas. Cada jugador puede quitar por turnos todos los palillos que quiera, pero de una sola fila. El jugador que quite el último palillo pierde. La aplicación debe estar preparada para que sea fácil de modificar el número de filas y el número de palillos por fila.

Planteamiento: Se divide el programa en tres clases: `FilaPalillos`, `JuegoPalillos` y `Main`.

a) La clase `FilaPalillos` trata con todo lo referente a una fila de palillos, en el constructor se recibe el número de palillos que contiene la fila. Además contiene los siguientes métodos:

- `quitaPalillos()`. Recibe el número de palillos que se quitan de la fila, si se pretenden quitar más palillos de los que hay devuelve `false`. Si no, los quita y devuelve `true`.
- `toString()`. Devuelve en forma de `String` tantos palillos como tenga la fila.
- `cuantosPalillos()`. Devuelve cuántos palillos tiene la fila.
- `añadePalillos()`. Recibe los palillos a añadir a la fila. No se utilizará en este programa, pero se añade para una posible reutilización para otro juego.

b) La clase `JuegoPalillos` trata con lo referente al juego, contiene un array de `FilaPalillos`. En el constructor se recibe un array que contiene en cada celda el número de palillos de cada fila, por tanto la longitud del array recibido es el número de filas que tiene el juego. Además contiene los siguientes métodos:

- `quitaPalillos()`. Recibe por parámetro la fila y el número de palillos que se quitan de esa fila. Si no se pueden quitar, bien porque no existe la fila, bien porque no tiene tantos palillos, devuelve `false`. Si no, se quitan y devuelve `true`.

- finDeJuego(). Devuelve true si no queda ningún palillo en ninguna fila.
- toString(). Devuelve en forma de String la información de los palillos que quedan en el juego. Para ello se utiliza el método toString() de FilaPalillos.

c) En la clase Main, se escribe la aplicación. Se crea un JuegoPalillos, en este caso con 7, 5 y 3 palillos en cada fila y posteriormente en un bucle, se muestra el juego, se pide al usuario la fila y el número de palillos que desea quitar mientras que se acabe el juego. Se controlan las posibles excepciones de entrada y no se cambia de turno si quitarPalillos() devuelve false.

Solución:

```
public class FilaPalillos{
    private int numPalillos;

    public FilaPalillos(int tamaño){
        numPalillos = tamaño;
    }

    public boolean quitaPalillos(int cuantos){
        if (cuantos > numPalillos){
            return false;
        }else{
            numPalillos -= cuantos;
            return true;
        }
    }
    public String toString(){
        String s = "";
        for (int i=0; i<numPalillos; i++){
            s += "|";
        }
        return s;
    }
    public void añadePalillos(int cuantos){
        numPalillos += cuantos;
    }
    public int cuantosPalillos(){
        return numPalillos;
    }
}
```

Se utiliza un String auxiliar para acumular palillos.

```
public class JuegoPalillos{
    private FilaPalillos[] filas;

    public JuegoPalillos(int[] palillos){
        filas = new FilaPalillos[palillos.length];
        for (int i = 0; i < filas.length; i++){
            filas[i] = new FilaPalillos(palillos[i]);
        }
    }

    public boolean quitaPalillos(int fila, int cuantos){
```

Cada celda del array es un objeto de tipo FilaPalillos que hay que crear.

```
        if (fila < 0 || fila >= filas.length)
            return false;
        else
            return filas[fila].quitaPalillos(cuantos);
    }

    public boolean finDeJuego(){
        for (int i = 0; i < filas.length; i++){
            if (filas[i].cuantosPalillos() != 0) return false;
        }
        return true;
    }
    public String toString(){
        String s = "";
        for (int i = 0; i < filas.length; i++){
            s += i + " " + filas[i] + "\n";
        }
        return s;
    }
}
```

En cuanto una fila no tenga 0 palillos, se devuelva `false`, si se sale del `for` es porque todas tienen 0.

Al método `toString()` lo llama automáticamente Java cuando necesita un `String`.

```
import java.util.Scanner;
import java.util.InputMismatchException;
public class Main{
    public static void main(String args[]){
        Scanner lector = new Scanner(System.in);
        int[] palillos  = {7,5,3};
        JuegoPalillos juego;
        String[] jugador = new String[2];
        jugador[0] = "Jugador 1";
        jugador[1] = "Jugador 2";
        int turno = 0;
        int fila;
        int cuantos;
        juego = new JuegoPalillos(palillos);

        do{
            try{
                System.out.println(juego);
                System.out.println(jugador[turno]+", elige fila");
                fila = lector.nextInt();
                System.out.println(jugador[turno]+", ¿cuántos palillos quieres quitar?");
                cuantos = lector.nextInt();
                if (juego.quitaPalillos(fila,cuantos)){
                    turno = (turno + 1) % 2;
                }else{
                    System.out.println("Introduce bien la fila y los palillos");
                }
            }catch (InputMismatchException e){
                System.out.println("Por favor introduce un número." );
                lector.next();
```

Para almacenar los nombres de los jugadores.

Para cambiar de turno.

```
            }catch (Exception exc){
                System.out.println("Se ha producido algún error " + exc.toString());
            }
        }while (!juego.finDeJuego());

        System.out.println("El ganador ha sido " + jugador[turno]);
    }
}
```

Comentario: Se pueden pedir los nombres a los usuarios al principio de la aplicación.

Si se quiere cambiar el número de filas o de palillos en cada fila, sólo hay que modificar el array palillos del `main`. Incluso se le puede pedir el número de filas y los palillos de cada fila al usuario.

Los métodos `quitarPalillos()` en lugar de ser `boolean`, podrían ser void y lanzar una excepción si no se pueden quitar los palillos. Se ha elegido esta implementación porque no se considera una situación excepcional que el usuario se confunda.

Ejercicio 5.4:

Escriba una clase Punto que complete la del Ejercicio 2.4 del Capítulo 2 para que, además de los constructores y los métodos getX(), getY() y distancia(), contenga también los siguientes métodos:

a) Dos métodos para modificar los valores de los atributos.
b) Un método toString() que devuelva la información del Punto de la siguiente manera (x,y).

Planteamiento: El método `toString()` concatena los paréntesis y comas como literales con los valores de los atributos, en el orden requerido.

Solución:

```
public class Punto{
    private double x;
    private double y;

    public Punto(double valorX, double valorY){
        x = valorX;
        y = valorY;
    }

    public Punto(){
        this(0,0);
    }

    public void setX(double valor){
        x = valor;
    }

    public void setY(double valor){
        y = valor;
    }

    public double getX(){
        return x;
    }
```

```
    public double getY(){
        return y;
    }

    public double distancia(Punto p){
        double distanciaX = p.getX() - x;
        double distanciaY = p.getY() - y;
        return Math.sqrt(distanciaX * distanciaX + distanciaY * distanciaY);
    }

    public String toString(){
        return "(" + x + "," + y + ")";
    }
}
```

Comentario: Fíjese que en el método distancia se utiliza el método `Math.sqrt()` para hacer la raíz cuadrada. Para elevar al cuadrado `distanciaX` y `distanciaY` se podría haber utilizado el método `Math.pow()`.

Ejercicio 5.5:

Utilizando la clase Punto del ejercicio anterior, escriba una clase Polígono. Esta clase tiene como atributo un array de objetos Punto. Se consideran adyacentes los objetos Punto que estén en celdas consecutivas del array y los puntos que están en la primera y última celda. Esta clase ha de tener los siguientes métodos:

a) el constructor, recibirá por parámetro un array con los objetos Punto que definen el Polígono.
b) trasladar(), recibe por parámetro el desplazamiento en la coordenada x y el desplazamiento en la coordenada y.
c) escalar(), recibe por parámetro el factor de escala para la coordenada x y el factor de escala para la coordenada y.
d) numVértices(), devuelve el número de vértices del Polígono.
e) toString(), devuelve la información de los puntos del Polígono, uno en cada línea.
f) perímetro(), devuelve el perímetro del Polígono.

Planteamiento:

a) En el constructor hay que controlar que el número de vértices del Polígono no es menor que 3. En caso contrario se lanza una excepción de tipo `IllegalArgumentException`.
b) Para trasladar el Polígono, se recorre el array desplazando cada uno de los puntos.
c) Para escalar se hace lo mismo, pero multiplicando su coordenada *x* por el factor *x* y su coordenada *y* por el factor *y*.
d) El número de vértices coincide con la longitud del array.
e) El método `toString()` llama al método `toString()` de cada uno de los puntos y los concatena en un `String` auxiliar que devuelve al terminar el método.
f) El perímetro se calcula utilizando el método `distancia()` de la clase `Punto`. Se calcula y suma la distancia de cada Punto con su Punto siguiente, sin olvidarse de la distancia entre el Punto de la celda 0 y el de la última celda.

Solución:

```
public class Polígono{

    private Punto[] vértices;

    public Polígono(Punto[] valor){
```

```
            if (valor.length < 3){
                throw new IllegalArgumentException();
            }
            vértices = valor;
        }
        public void escalar(double x, double y){
            double valX, valY;
            for (Punto p  :  vértices){
                valX = p.getX();
                valY = p.getY();
                p.setX(valX * x);
                p.setY(valY * y);
            }
        }
        public void trasladar(double x, double y){
            double valX, valY;
            for (Punto p  :  vértices){
                valX = p.getX();
                valY = p.getY();
                p.setX(valX + x);
                p.setY(valY + y);
            }
        }
        public int numVértices(){
            return vértices.length;
        }
        public double perímetro(){
            double acumulador = 0;
            for (int i = 0; i < vértices.length - 1; i++){
                acumulador += vértices[i].distancia(vértices[i + 1]);
            }
            acumulador += vértices[0].distancia(vértices[vértices.length - 1]);
            return acumulador;
        }

        public String toString(){
            String cadena="";
            for (Punto p  :  vértices){
                Cadena += p + "\n";
            }
            return cadena;
        }
    }
```

IllegalArgumentException hereda de RuntimeException, por eso no es necesario indicarlo en la cabecera del método.

Comentario: El método escalar(), tal y como está definido, además de escalar, traslada. Por ejemplo, un Polígono formado por los vértices (1,1), (1,2), (2,2), (2,1) al escalarlo por 2 en el eje *x* y por 3 en el eje *y* pasaría a estar en el (2,3), (2,6), (4,6), (4,3), con lo que se ha escalado (es el doble de ancho y el triple de alto), pero también se ha trasladado.

Ejercicio 5.6:

Escriba una aplicación en la que:

a) *Cree un Polígono de cuatro vértices, que serán (0,0), (2,0), (2,2), (0,2).*
b) *Muestre la información del polígono y su perímetro por pantalla.*
c) *Traslade el polígono 4 en el eje x y –3 en el eje y.*
d) *Muestre la información del polígono y su perímetro por pantalla.*
e) *Escale el polígono por 3 en el eje x y por 0,5 en el eje y.*
f) *Muestre la información del polígono y su perímetro por pantalla.*

Planteamiento: Para crear un Polígono se necesita un array de objetos Punto, por lo que lo primero que hay que hacer es crear este array y rellenarlo con los cuatro puntos.

Posteriormente se creará el objeto de la clase Polígono y se llamará a cada uno de los métodos indicados. Para mostrar a pantalla la información se utilizará el método `toString()`.

Solución:

```
public class PruebaPolígono {
    public static void main(String args[]) {
        Punto[] vert = {new Punto(0,0), new Punto(2,0), new Punto(2,2), new Punto(0,2)};
        Polígono miPolígono;
        miPolígono = new Polígono(vert);
        System.out.println(miPolígono);
        System.out.println("Perímetro: " + miPolígono.perímetro());
        miPolígono.trasladar(4, -3);
        System.out.println(miPolígono);
        System.out.println("Perímetro: " + miPolígono.perímetro());
        miPolígono.escalar(3, 0.5);
        System.out.println(miPolígono);
        System.out.println("Perímetro: " + miPolígono.perímetro());
    }
}
```

Declaración, creación e inicialización del array en una sola sentencia.

Cuando se necesita un `String`, se llama automáticamente al método `toString()`.

Herencia

Ejercicio 5.7:

Escriba una clase PolígonoColoreado que herede de la clase Polígono del Ejercicio 5.5. En esta clase se debe añadir un atributo color y métodos para cambiar y obtener el color. El método toString() se tiene que modificar para que devuelva el color del Polígono además de los vértices que lo componen.

Planteamiento: La clase `PolígonoColoreado` hereda de la clase `Polígono`. Tiene que declarar un atributo para representar el color. El color se declara de la clase `Color` del paquete `awt`. Se tienen que implementar los métodos para cambiar y devolver el color.

Además, se sobreescribe el método `toString()`. Este método añade la información del color y llama al método `toString()` de la superclase.

El constructor recibe por parámetro el array de puntos. Lo primero que hay que hacer es llamar al constructor de la superclase. Posteriormente, se da valor al atributo color, valor que también se recibe por parámetro.

Solución:

```
import java.awt.Color;
public class PolígonoColoreado extends Polígono{
    private Color color;
    public PolígonoColoreado(Punto[] vértices, Color color) {
        super(vértices);
        this.color = color;
```

Hereda de `Polígono`.

Llamada al constructor de la superclase.

```
    }
    public void setColor(Color color){
        this.color = color;
    }
    public Color getColor(){
        return color;
    }
    public String toString(){
        return "Polígono con color " + color + "\n"+super.toString();
    }

}
```

Llamada al método toString() de la superclase.

Comentario: El método toString() de la clase Color del paquete java.awt, al escribirlo, devuelve algo con el siguiente aspecto para el color negro: java.awt.Color[r=0,g=0,b=0]. Aunque no sea muy fácil de entender, es preferible utilizar esta clase por si en el futuro se tiene la oportunidad de representarla gráficamente, ya que las clases gráficas de Java utilizan objetos de este tipo para representar los colores.

También se podría haber elegido declarar Color como un tipo enumerado y trabajar con dicho tipo.

Ejercicio 5.8:

Escriba una aplicación en la que:

a) *Cree un PolígonoColoreado de cuatro vértices que serán (0,0), (2,0), (2,2), (0,2) y de color rojo.*
b) *Muestre la información del polígono y su perímetro por pantalla.*
c) *Traslade el polígono 4 en el eje x y –3 en el eje y.*
d) *Cambie el color del PolígonoColoreado a azul.*
e) *Muestre la información del polígono por pantalla.*

Planteamiento: Para crear un PolígonoColoreado se necesita un array de puntos, por lo que lo primero que hay que hacer es crear este array y rellenarlo con los cuatro puntos.

Posteriormente, se crea el objeto de la clase PolígonoColoreado. Como segundo parámetro se pasa Color.RED que es la constante para el color rojo de la clase Color.

La clase PolígonoColoreado contiene todos los métodos de la clase Polígono, por lo que se puede invocar a los métodos perímetro() y trasladar() como si estuviesen declarados en la clase PolígonoColoreado. Para mostrar a pantalla la información se utilizará el método toString(). En este caso Java ejecutará el método toString() de PolígonoColoreado, ya que está sobreescrito.

Solución:

```
import java.awt.Color;
public class PruebaPolígonoColoreado {
    public static void main(String args[]) {
        Punto[] vert = {new Punto(0, 0), new Punto(2, 0), new Punto(2, 2), new Punto(0, 2)};
        PolígonoColoreado miPolígono;
        miPolígono = new PolígonoColoreado(vert, Color.RED);
        System.out.println(miPolígono);
        System.out.println("Perímetro: " + miPolígono.perímetro());
        miPolígono.trasladar(4, -3);
        miPolígono.setColor(Color.BLUE);
        System.out.println(miPolígono);
    }
}
```

Ejercicio 5.9:

Escriba una clase Multimedia para almacenar información de objetos de tipo multimedia (películas, discos, mp3...). Esta clase contiene título, autor, formato y duración como atributos. El formato puede ser uno de los siguientes: wav, mp3, midi, avi, mov, mpg, cdAudio y dvd. El valor de todos los atributos se pasa por parámetro en el momento de crear el objeto. Esta clase tiene, además, un método para devolver cada uno de los atributos y un método toString() que devuelve la información del objeto. Por último un método equals() que recibe un objeto de tipo Multimedia y devuelve true en caso de que el título y el autor sean iguales a los del objeto que llama al método y false en caso contrario.

Planteamiento: Para el formato de los objetos Multimedia, se debe crear un tipo enumerado de nombre `Formato` con los valores indicados anteriormente. Este tipo enumerado se crea en un archivo aparte para que pueda ser utilizado por otras clases.

Solución:

```
public enum Formato {wav, mp3, midi, avi, mov, mpg, cdAudio, dvd}

public class Multimedia {
    private String título;
    private String autor;
    private Formato formato;
    private double duración;

    public Multimedia(String título, String autor, Formato formato, double duración) {
      this.título   = título;
      this.autor    = autor;
      this.formato  = formato;
      this.duración = duración;
    }

    public String getTítulo(){
      return título;
    }

    public String getAutor(){
      return autor;
    }

    public Formato getFormato(){
      return formato;
    }

    public double getDuración(){
      return duración;
    }

    public String toString(){
      return "Título: " + título + " De: " + autor + "\n" +
             "Formato: " + formato + " Duración: " + duración;
    }

    public boolean equals(Multimedia m){
```

```
            return título.equals(m.getTítulo()) && autor.equals(m.getAutor());
        }
    }
```

Ejercicio 5.10:

Escriba una clase Película que herede de la clase Multimedia anterior. La clase Película tiene, además de los atributos heredados, un actor principal y una actriz principal. Se permite que uno de los dos sea nulo, pero no los dos. La clase debe tener dos métodos para obtener los dos nuevos atributos y debe sobreescribir el método toString() para que devuelva, además de la información heredada, la nueva información.

Planteamiento: La clase `Película` hereda de `Multimedia`.

En el constructor se reciben los valores de los atributos de la superclase y los dos nuevos. Se debe controlar que los valores de los dos nuevos atributos (actor y actriz principal) no sean ambos nulos. En el caso de que ambos sean nulos se lanzará una excepción `IllegalArgumentException`.

Se sobreescribe el método `toString()` y se añaden los dos métodos nuevos.

Solucion:

```
public class Película extends Multimedia{

    private String actorPrincipal;
    private String actrizPrincipal;

    public Película(String título, String autor, Formato formato, double duración, String actor,
        String actriz) {
        super(título, autor, formato, duración); ← Llamada al constructor de la superclase.
        if (actor == null && actriz == null){
            throw new IllegalArgumentException("Tiene que haber al menos un protagonista.");
        }
        actorPrincipal  = actor;
        actrizPrincipal = actriz;
    }

    public String getActor(){
        return actorPrincipal;
    }

    public String getActriz(){
        return actrizPrincipal;
    }

    public String toString(){
        String s = "Protagonizado por ";
        if (actrizPrincipal != null){
            s += actrizPrincipal;
            if (actorPrincipal != null){
                s += " y " + actorPrincipal;
            }
        }else{
            assert actorPrincipal != null; ← No puede ocurrir que actrizPrincipal
                                             y actorPrincipal sean nulos a la vez.
            s += actorPrincipal;
        }
```

```
        return super.toString() + "\n" + s;
    }
}
```

Llamada al método toString() de la superclase.

Nota: Las aserciones se han explicado en el Capítulo 4.

Ejercicio 5.11:

Escriba una clase ListaMultimedia para almacenar objetos de tipo multimedia. La clase debe tener un atributo que sea un array de objetos Multimedia y un entero para contar cuántos objetos hay almacenados. Además, tiene un constructor y los siguientes métodos:

a) el constructor recibe por parámetro un entero indicando el número máximo de objetos que va a almacenar.
b) int size(): devuelve el número de objetos que hay en la lista.
c) boolean add(Multimedia m): añade el objeto al final de la lista y devuelve true, en caso de que la lista esté llena, devolverá false.
d) Multimedia get(int posición): devuelve el objeto situado en la posición especificada.
e) int indexOf(Multimedia m): devuelve la posición del objeto que se introduce por parámetro, si no se encuentra, devolverá -1.
f) String toString() devuelve la información de los objetos que están en la lista.

Planteamiento:

a) En el constructor se crea el array con el tamaño especificado y se inicializa el contador a 0. Se implementa un método `llena()` para uso interno, por lo que es un método privado.
b) El número de objetos se almacena en el contador, así que simplemente se devuelve el valor del atributo.
c) Este método se usa en `add()`, donde en el caso de que no esté llena, se almacena el objeto en la celda indicada por contador y se incrementa éste.
d) En `get()` se comprueba que la posición sea válida y se devuelve el objeto. Si no fuese una posición válida, se lanzaría una excepción de tipo `IndexOutOfBoundsException`.
e) En `indexOf()` se busca el elemento recibido por parámetro. Para comparar el parámetro con los objetos de la lista se utiliza el método `equals()` de Multimedia.
f) En el método `toString()` se utiliza un `String` para ir acumulando las llamadas al método `toString()` de los objetos multimedia. Finalmente se devuelve el `String`.

Solución:

```
public class ListaMultimedia {
    private Multimedia[] lista;
    private int contador;

public ListaMultimedia(int tamañoMáximo) {
        lista    = new Multimedia[tamañoMáximo];
        contador = 0;
    }

    public int size(){
        return contador;
    }

    private boolean llena(){
        return contador == lista.length;
    }
```

Método privado para que no se pueda usar desde fuera de la clase.

```
    public boolean add(Multimedia m){
        if (llena()){
            return false;
        }else{
            lista[contador] = m;
            contador++;
            return true;
        }
    }

    public Multimedia get(int posición){
        if (posición < 0 || posición >= contador)
            throw new IndexOutOfBoundsException();
        return lista[posición];
    }

    public int indexOf(Multimedia m){
        for (int i=0; i<contador; i++){  <---- Sólo se recorre hasta la celda que indique contador.
            if(m.equals(lista[i])){
                return i;
            }
        }
        return -1;  <---- Esta sentencia sólo se ejecuta si no se cumple nunca la condición del if.
    }

    public String toString(){
        String s="";
        for (int i=0; i<contador; i++){
            s += lista[i].toString() + "\n\n";
        }
        return s;
    }
}
```

Si `lista[i]` referencia a una Película, por ligadura dinámica llamará a `toString()` de Película.

Comentario: La clase ListaMultimedia no es completa. Se necesitaría, por ejemplo, un método para poder eliminar objetos multimedia de ella. Por no complicar más el ejercicio se ha decidido dejarla así.

Para el método `add()` se ha decidido que si no se puede añadir el elemento se devuelve `false` y si se añade se devuelve `true`. Se ha optado por esta opción porque el hecho de que la lista esté llena no se considera una situación excepcional, en el caso de que se considerase que sí es una situación excepcional se podría lanzar una excepción y declarar el método `void` de la siguiente manera: `public void add(Multimedia m) throws ListaLlenaException`, donde `ListaLlenaException` sería un tipo de excepción declarado previamente.

Ejercicio 5.12:

Escriba una aplicación donde:

a) *Se cree un objeto de tipo ListaMultimedia de tamaño máximo 10.*
b) *Se creen tres películas y se añadan a la lista.*
c) *Se muestre la lista por pantalla.*
d) *Se cree un objeto de tipo Película introduciendo el título y el autor de una de las películas de la lista, para el resto de los argumentos se utilizan valores no significativos.*
e) *Busque la posición de este objeto en la lista.*

f) *Obtenga el objeto que está en esa posición y lo muestre por pantalla junto con la posición en la que se encuentra.*

Planteamiento: Para mostrar la lista por pantalla se utiliza el método `toString()`.

Cuando se crea el objeto `Película` de la opción d) para buscar una película, algunos argumentos no importan en la comparación, por lo que se podría utilizar `null` para los `String` y para el `Formato`, y 0 para el valor `double`. Pero existe un problema, en el constructor de película no se permite que el actor y la actriz principal sean `null` al mismo tiempo, por lo que se pasarán cadenas vacías.

El método `indexOf()` para buscar el objeto utiliza `equals()` de `Multimedia`, que sólo compara esos dos atributos (`título` y `autor`) por lo que se devuelve la posición. Al llamar al método `get()` con esa posición se obtiene el objeto completo.

Solución:

```
public class Main {

    public static void main(String[] args) {
        ListaMultimedia lista = new ListaMultimedia(10);
        Película peli;
        int posición;
        lista.add(new Película("Million dollar baby",
                               "Clint Eastwood",
                               Formato.dvd,
                               137,
                               "Clint Eastwood",
                               "Hillary Swank"));
        lista.add(new Película("El aviador",
                               "Martin Scorsese",
                               Formato.dvd,
                               168,
                               "Leonardo di Caprio",
                               null));
        lista.add(new Película("Mar adentro",
                               "Alejandro Amenábar",
                               Formato.avi,
                               125,
                               "Javier Bardem",
                               "Belén Rueda"));

        System.out.println(lista.toString());

        peli = new Película("Mar adentro", "Alejandro Amenábar", null, 0, "", "");
        posición = lista.indexOf(peli);
        System.out.println("Posición " + posición + "\n" + lista.get(posición));
    }
}
```

El método add() recibe objetos de tipo Multimedia, como Película hereda de Multimedia, son aceptados también.

Se crea el objeto con el título y el autor, el resto de los argumentos no interesan.

Imprimirá todos los datos de «Mar adentro».

Nota: Fíjese en que `lista.toString()` llama a los métodos `toString()` de `Película`, por ligadura dinámica.

Ejercicio 5.13:

Escriba una clase Disco que herede de la clase Multimedia ya realizada. La clase Disco tiene, aparte de los elementos heredados, un atributo para almacenar el género al que pertenece (rock, pop, funk, etc.). La clase

debe tener un método para obtener el nuevo atributo y debe sobreescribir el método toString() para que devuelva, además de la información heredada, la nueva información.

Planteamiento: El género se define de un tipo enumerado Género, que se implementa en un archivo aparte para que pueda ser usado por el resto de las clases. La clase Disco hereda de Multimedia y añade el nuevo atributo de tipo Género.

Cuando sobreescriba el método toString() debe llamar al método toString() de la superclase y concatenar la parte nueva.

Solución:

```
public enum Género {rock, pop, soul, funky, flamenco, clásica, blues,
                tecno, ambient, punk, jazz, hiphop, bso};

public class Disco extends Multimedia{
    private Género género;

    public Disco(String título, String autor, Formato formato, double duración, Género género){
        super(título, autor, formato, duración);
        this.género = género;
    }

    public Género getGénero(){
        return género;
    }

    public String toString(){
        return super.toString() + "\nGénero: " + género;
    }
}
```

Enumerado de tipo Género, aquí podemos añadir nuevos tipos de géneros musicales.

Llamada al constructor de la superclase.

Llamada al método toString() de la superclase.

Ejercicio 5.14:

Escriba una aplicación donde:

- *a) Se cree un objeto de tipo ListaMultimedia de tamaño máximo 10.*
- *b) Se creen tres discos y se añadan a la lista.*
- *c) Se muestre la lista por pantalla.*
- *d) Se cree un objeto de tipo Disco introduciendo el título y el autor de uno de los discos de la lista, para el resto de los argumentos se utilizan valores no significativos.*
- *e) Busque la posición de este objeto en la lista.*
- *f) Obtenga el objeto que está en esa posición y lo muestre por pantalla junto con la posición en la que se encuentra.*

Planteamiento: Para mostrar la lista por pantalla se utiliza el método toString().

Cuando se crea el objeto Disco de la opción d) para los argumentos no significativos se podría utilizar null para el Formato y para el Género, y 0 para el valor double.

El método indexOf() para buscar el objeto utiliza equals() de Multimedia, que sólo compara esos dos atributos (título y autor) por lo que se devuelve la posición sin problema. Al llamar al método get() con esa posición se obtiene el objeto completo.

Solución:

```
public class AppListaDiscos {
```

```
        public static void main(String args[]){
            ListaMultimedia lista = new ListaMultimedia(10);
            Disco disco;
            int posición;
            lista.add(new Disco("Hopes and Fears", "Keane", Formato.mp3, 45, Género.pop));
            lista.add(new Disco("How to dismantle an atomic bomb", "U2",
                             Formato.cdAudio, 49, Género.rock));
            lista.add(new Disco("Soy gitano", "Camarón", Formato.cdAudio, 32, Género.flamenco));
            System.out.println(lista.toString());
            disco = new Disco("Soy gitano", "Camarón", null, 0, null);
            posición = lista.indexOf(disco);
            System.out.println("Posición " + posición + "\n" + lista.get(posición));
        }
}
```

Se pasan los Discos igual que antes se pasaban las películas.

Llamará automáticamente al método toString() de Disco.

Comentario: Fíjese que la misma clase `ListaMultimedia` que se utilizó antes en el Ejercicio 5.12 para almacenar objetos de tipo `Película` se utiliza ahora para almacenar objetos de tipo `Disco`. Como tanto `Disco` como `Película` heredan de `Multimedia`, se pueden almacenar en una `ListaMultimedia`, con lo que se consigue una clase más reutilizable que si se hubiese implementado una lista de `Películas`.

Nota: Fíjese en que `lista.toString()` llama a los métodos `toString()` de `Disco`, por ligadura dinámica.

ÁMBITOS Y VISIBILIDAD

Ejercicio 5.15:

a) *Escriba una clase Coche de la que van a heredar CocheCambioManual y CocheCambioAutomatico. Los atributos de los coches son la matrícula, la velocidad y la marcha. Para este ejercicio no se permite la marcha atrás, por tanto no se permiten ni velocidad negativa, ni marcha negativa. En el constructor se recibe el valor de la matrícula por parámetro y se inicializa el valor de la velocidad y la marcha a 0. Además tendrá los siguientes métodos:*

- *getMatrícula(): que devuelve el valor de la matrícula.*
- *getMarcha(): devuelve el valor de la marcha.*
- *getVelocidad(): devuelve el valor de la velocidad.*
- *acelerar(): recibe por parámetro un valor al acelerar el coche.*
- *frenar(): recible por parámetro un valor al frenar el coche.*
- *toString(): devuelve en forma de String la información del coche.*
- *cambiaMarcha: recibe por parámetro la marcha a la que se tiene que cambiar el coche. Este método es protected, para que puedan acceder a él las clases que heredan de Coche, pero no las clases de otros paquetes.*

b) *La clase CocheCambioManual sobreescribe el método cambiaMarcha() y lo hace público, para que pueda ser llamado desde cualquier clase. No permite que se cambie a una marcha negativa.*

c) *La clase CocheCambioAutomatico sobreescribe los métodos acelerar() y frenar() para que cambie automáticamente de marcha conforme se va acelerando y frenando.*

Planteamiento:

a) La clase `Coche` tiene tres atributos: matrícula de tipo `String`, velocidad de tipo `double` y marcha de tipo `int`. Los tres se declaran con modificador de acceso privado para que no se pueda acceder a ellos desde otras clases. El método `cambiaMarcha()` se declara `protected` porque se necesita acceder a él desde las subclases, pero no es un método que vayan a tener todos los tipos de coche.

El resto de los métodos y el constructor se declaran públicos, ya que son métodos a los que se puede acceder desde cualquier clase.

En el método acelerar() si se recibe un parámetro negativo se llama a frenar() con el parámetro en positivo. Si es positivo se añade a la velocidad el valor recibido por parámetro.

En el método frenar() si se recibe un parámetro negativo se llama a acelerar() con el parámetro en positivo. Si es positivo se sustrae a la velocidad el valor recibido por parámetro. Si se queda con una cantidad negativa se le da el valor de 0.

b) La clase CocheCambioManual recibe en el constructor la matrícula que pasa al constructor de la superclase. Sobreescribe el método cambiaMarcha() y le da acceso público ya que, para un coche con cambio manual, debe ser un método accesible desde cualquier clase. En el caso de que reciba por parámetro una cantidad negativa lanza una excepción IllegalArgumentException. Si no es así, para cambiar a otra marcha llama al método cambiaMarcha() de la superclase, ya que al tener el atributo marcha el modificador de acceso privado no se puede acceder al atributo directamente.

Este cambio de visibilidad se permite porque el método se está haciendo más accesible, nunca se permitiría que se hiciera menos accesible.

c) La clase CocheCambioAutomatico recibe en el constructor la matrícula que pasa al constructor de la superclase.

Sobreescribe los métodos acelerar() y frenar(), según se va acelerando o frenando se cambia la marcha. Lo primero que se hace es llamar al método sobreescrito, después se obtiene la velocidad llamando al método getVelocidad() heredado, ya que el atributo tiene modificador de acceso privado. Dependiendo de cuál sea la velocidad se cambia la marcha. Para ello se utiliza el método cambiaMarcha() que es protected y por tanto puede ser accedido desde la subclase. Al atributo no se podría acceder directamente porque es privado.

Solución:

```
public class Coche {

    private String matrícula;
    private double velocidad;
    private int marcha;
    public Coche(String matrícula) {
        this.matrícula = matrícula;
        velocidad = 0;
        marcha    = 0;
    }

    public String getMatrícula(){
        return matrícula;
    }

    public void acelerar(int cuanto){
        if (cuanto < 0){
            frenar(-cuanto);  <--- Si el parámetro es negativo se
                                   cambia el signo y se llama a frenar().
        }else{
            velocidad += cuanto;
        }
    }

    public void frenar(int cuanto){
        if (cuanto < 0){
            acelerar (-cuanto);
```

```
        }else{
            velocidad -= cuanto;
            if (velocidad < 0){
                velocidad = 0;   <-- Si el parámetro es mayor que la velocidad, ésta se pone a 0.
            }
        }
    }

    protected void cambiaMarcha(int marcha){
        this.marcha = marcha;
    }

    public int getMarcha(){
        return marcha;
    }

    public double getVelocidad(){
        return velocidad;
    }

    public String toString(){
        return "Matrícula: " + matrícula +
              "\nVelocidad: " + velocidad +
              "\nMarcha:    " + marcha;
    }
}

public class CocheCambioManual extends Coche{

    public CocheCambioManual(String matrícula) {
        super(matrícula);   <-- Llamada al constructor de la superclase.
    }

    public void cambiaMarcha(int marcha){   <-- Se sobreescribe con un modificador de acceso público.
        if (marcha < 0){
            throw new IllegalArgumentException();

        super.cambiaMarcha(marcha);
    }
}

public class CocheCambioAutomatico extends Coche{

    public CocheCambioAutomatico(String matrícula) {
        super(matrícula);
    }

    public void acelerar(int cuanto){
        super.acelerar(cuanto);   <-- Se llama al método acelerar() de la superclase.
        if (getVelocidad() < 10){
            cambiaMarcha(1);
```

```
        }else if (getVelocidad() < 30){
            cambiaMarcha(2);
        }else if (getVelocidad() < 50){
            cambiaMarcha(3);
        }else if (getVelocidad() < 80){
            cambiaMarcha(4);
        }else{
            cambiaMarcha(5);
        }
    }

    public void frenar(int cuanto){
        super.frenar(cuanto);
        if (getVelocidad() < 5){
            cambiaMarcha(1);
        }else if (getVelocidad() < 20){
            cambiaMarcha(2);
        }else if (getVelocidad() < 40){
            cambiaMarcha(3);
        }else if (getVelocidad() < 60){
            cambiaMarcha(4);
        }else{
            cambiaMarcha(5);
        }
    }
}
```

Nota: Se podría haber permitido velocidades negativas y la marcha atrás, pero el ejercicio se complica y no aporta nada nuevo en lo que se refiere a herencia, ámbito y visibilidad.

Compatibilidad de tipos

Ejercicio 5.16:

Escriba una aplicación que pida por teclado la matrícula de un coche y pregunte si el coche es con cambio automático o no. Posteriormente, debe crear un coche con las características indicadas por el usuario y mostrarlo. Acelerar el coche en 60 km/h, si es un coche con cambio manual, cambiar la marcha a tercera y volverlo a mostrar.

Planteamiento: Para leer por teclado se utiliza la clase `Scanner`. Se declara un objeto de tipo `Coche`, superclase de las otras dos que nos sirve para almacenar tanto una referencia a un `CocheCambioAutomatico` como a un `CocheCambioManual`. Se almacena la matrícula en una variable, y la respuesta a si tiene cambio automático o no en otra. De esta última, se compara el primer carácter con `'s'` o `'S'` para permitir mayúsculas.

Dependiendo de la respuesta del usuario se creará un objeto `CocheCambioAutomatico` o un `CocheCambioManual`, cualquiera de ellos se guardará en la referencia declarada de tipo `Coche`, realizándose así un cambio de tipo a la superclase, que se realiza automáticamente.

Para acelerar el coche, se llama al método `acelerar()`. En caso de que se esté referenciando a un `CocheCambioAutomatico`, por ligadura dinámica se llama al método sobreescrito.

Si se trata de un `CocheCambioManual` se debe llamar al método `cambiaMarcha()` para poner la tercera. Para comprobar que realmente se trata de un `CocheCambioManual` se utiliza el operador `instanceof`, éste nos dice si es posible la conversión de tipo a la subclase, en caso de que así sea, todavía se debe hacer esta conversión de tipo explícitamente.

Para mostrar el coche por pantalla se utiliza el método `toString()` para que nos devuelva la información a mostrar. No es necesario llamar explícitamente a este método ya que Java lo llama automáticamente cuando detecta que se necesita un `String` en lugar de un objeto de otro tipo.

Solución:

```
import java.util.*;
public class Main {

    public static void main(String[] args) {
        Scanner lector = new Scanner(System.in);
        Coche coche;
        String matrícula;
        String sino;
        System.out.print("¿Cuál es la matrícula? ");
        matrícula = lector.nextLine();
        System.out.print("¿Tiene cambio automático? (s/n) ");
        sino = lector.nextLine();
        if (sino.charAt(0) == 's' || sino.charAt(0) == 'S'){  // Sólo se compara el primer carácter de la cadena.
            coche = new CocheCambioAutomatico(matrícula);
        }else{
            coche = new CocheCambioManual(matrícula);  // Conversión de tipo a la superclase implícita.
        }

        System.out.println(coche);
        coche.acelerar(60);
        if (coche instanceof CocheCambioManual){
            ((CocheCambioManual)coche).cambiaMarcha(3);  // Conversión de tipo a la subclase explícita.
        }

        System.out.println(coche);
    }
}
```

Comentario: El hecho de que el modificador de acceso protected permita el acceso dentro del mismo paquete, consigue que no sea necesaria la conversión explícita a la subclase para llamar a `cambiaMarcha()`, siempre que la clase `Main` esté en el mismo paquete que `Coche`. Si la clase `Main` está en un paquete distinto, la conversión de tipo a la subclase es obligatoria. Si no, no compilaría.

POLIMORFISMO

Ejercicio 5.17:

Escriba una aplicación donde:

a) *Se cree un objeto de tipo ListaMultimedia de tamaño máximo 10. ListaMultimedia del Ejercicio 5.11.*
b) *Se creen tres discos y se añadan a la lista. Disco del Ejercicio 5.13.*
c) *Se creen tres películas y se añadan a la lista. Película del Ejercicio 5.10.*
d) *Trabajando con la lista y suponiendo que no se sabe en qué posiciones están los discos y las películas:*
   1. *Se muestre la lista por pantalla.*
   2. *Se calcule la duración total de los objetos de la ListaMultimedia.*
   3. *Se muestre cuántos discos hay de rock.*
   4. *Se obtenga cuántas películas no tienen actriz principal.*

Planteamiento: Se crea un objeto de tipo ListaMultimedia como en ejercicios anteriores.
Se añaden los elementos como en ejercicios anteriores.

1) Para mostrar la lista por pantalla se utiliza el método toString() de ListaMultimedia, que por ligadura dinámica llama al método toString() de Película o Disco según se referencie a una Película o un Disco.
2) Para calcular la duración total de los objetos Multimedia, se recorre la ListaMultimedia hasta tamaño(), con el método get() se obtiene cada uno de los objetos y se obtiene su duración con el método getDuración() de Multimedia.
3) Para contar cuántos discos hay de rock, se recorre la ListaMultimedia hasta tamaño(), con el método get() se obtiene cada uno de los objetos y con el operador instanceof se comprueba si es un Disco. Si se cumple esta condición, se transforma el objeto a Disco y se le pregunta su género con el método getGénero().
4) Para contar cuántas películas hay sin actriz principal, se recorre la ListaMultimedia hasta tamaño(), con el método get() se obtiene cada uno de los objetos y con el operador instanceof se comprueba si es una Película. Si se cumple esta condición, se transforma el objeto a Película y se comprueba si su actriz principal es null utilizando el método getActriz().

Solución:

```
public class Main {
    public static void main(String args[]){
        // crear la lista
        ListaMultimedia lista = new ListaMultimedia(10);
        // añadir tres discos
        lista.add(new Disco("Hopes and Fears", "Keane", Formato.mp3, 45, Género.pop));
        lista.add(new Disco("How to dismantle an atomic bomb", "U2",
                    Formato.cdAudio, 49, Género.rock));
        lista.add(new Disco("Soy gitano", "Camarón", Formato.cdAudio, 32, Género.flamenco));
        // añadir tres películas
        lista.add(new Película("Million dollar baby",
                            "Clint Eastwood",
                            Formato.dvd,
                            137,
                            "Clint Eastwood",
                            "Hillary Swank"));
        lista.add(new Película("El aviador",
                            "Martin Scorsese",
                            Formato.dvd,
                            168,
                            "Leonardo di Caprio",
                            null));
        lista.add(new Película("Mar adentro",
                            "Alejandro Amenábar",
                            Formato.avi,
                            125,
                            "Javier Bardem",
                            "Belén Rueda"));
        // mostrar la lista
        System.out.println(lista.toString());
        double duraciónTotal = 0;
        for (int i=0; i< lista.tamaño(); i++){
            duraciónTotal += lista.get(i).getDuración();
        }
```

Llamará automáticamente al método toString() de Disco o Película, según corresponda.

Los objetos Película y Disco tienen el método getDuración() heredado de Multimedia.

```
        System.out.println("La duración de toda la lista es: " + duraciónTotal);

        // calcular cuántos discos de rock
        int cuantosRock = 0;
        for (int i = 0; i < lista.tamaño(); i++){
            if (lista.get(i) instanceof Disco){      <-- Comprueba que es de tipo Disco.
                Disco disco = (Disco) lista.get(i);  <-- Convierte a Disco para poder llamar a getGénero().
                if (disco.getGénero() == Género.rock){
                    cuantosRock++;
                }
            }
        }
        System.out.println("Hay " + cuantosRock + " discos de rock.");

        // calcular cuántas películas sin actriz
        int sinActriz = 0;
        for (int i = 0; i < lista.tamaño(); i++){
            if (lista.get(i) instanceof Película){   <-- Comprueba que es de tipo Película.
                Película peli = (Película) lista.get(i);  <-- Convierte a Película para poder llamar a getActriz().
                if (peli.getActriz() == null){
                    sinActriz++;
                }
            }
        }
        System.out.println("Hay " + sinActriz + " películas sin actriz principal.");
    }

}
```

Ejercicio 5.18:

Se va a implementar un simulador de Vehículos. Existen dos tipos de Vehículo: Coche y Camión.

a) Sus características comunes son la matrícula y la velocidad. En el momento de crearlos, la matrícula se recibe por parámetro y la velocidad se inicializa a 0. El método toString() de los vehículos devuelve información acerca de la matrícula y la velocidad. Además se pueden acelerar, pasando por parámetro la cantidad en km/h que se tiene que acelerar.

b) Los coches tienen además un atributo para el número de puertas, que se recibe también por parámetro en el momento de crearlo. Tiene además un método que devuelve el número de puertas.

c) Los camiones tienen un atributo de tipo Remolque que inicializa a null (para indicar que no tiene remolque). Además tiene un método ponRemolque(), que recibe el Remolque por parámetro, y otro quitaRemolque(). Cuando se muestre la información de un camión que lleve remolque, además de la matrícula y velocidad del camión, debe aparecer la información del remolque.

d) En esta clase hay que sobreescribir el método acelerar de manera que si el camión tiene remolque y la velocidad más la aceleración superan los 100 km/h se lance una excepción de tipo DemasiadoRapidoException.

e) Hay que implementar la clase Remolque. Esta clase tiene un atributo de tipo entero que es el peso y cuyo valor se le da en el momento de crear el objeto. Debe tener un método toString() que devuelva la información del remolque.

f) Implementar la clase DemasiadoRapidoException.

Implementar este diseño con la estructura más conveniente.

Planteamiento: Se va a implementar una clase Vehículo que sirve de superclase para Coche y Camión. Un Remolque no es un Vehículo, así que se implementa sin heredar de Vehículo.

a) En esta clase Vehículo se declara e implementa la parte común entre Coche y Camión, es decir:
   - los atributos matrícula y velocidad,
   - el constructor que inicializa los dos atributos,
   - el método `acelerar()` y
   - el método `toString()`

   Posteriormente se implementan las clases Coche y Camión que heredan de Vehículo.

b) En Coche se añade el atributo `numPuertas` y el método para devolverlo. En el constructor se llama al constructor de la superclase.

c) En Camión se añade el atributo `remolque` de tipo `Remolque` y los métodos para poner y quitar el remolque, además se sobreescribe el método `toString()` para mostrar la información del remolque si lo tiene.

d) Al sobreescribir el método `acelerar()` se lanza una excepción de tipo `DemasiadoRapidoException` que hay que indicar en la cabecera mediante una cláusula `throws`. Java no permite, en un método que sobreescribe a otro, lanzar una excepción que no se está lanzando en el método sobreescrito. Por este motivo en el método `acelerar()` de Vehículo se debe indicar en la cláusula `throws` que se pueden lanzar este tipo de excepciones, aunque no se esté lanzando.

e) La clase Remolque no hereda de Vehículo. La clase Camión tiene una relación de composición con Remolque.

f) La clase `DemasiadoRapidoException` hereda de `Exception`.

Solución:

```
public class Vehiculo {
    private String matrícula;
    private double velocidad;

    public Vehiculo(String mat) {
        matrícula = mat;
        velocidad = 0;
    }

    public double getVelocidad(){
        return velocidad;
    }

    public void acelerar(double cantidad)throws DemasiadoRapidoException{
        velocidad += cantidad;
    }

    public String toString(){
        return "El vehículo con matrícula " + matrícula + " va a " + velocidad + " km/h";
    }
}
```

Indica que lanza la excepción para que los métodos de la subclase la puedan lanzar.

```
public class Coche extends Vehiculo{
    private int numPuertas;

    public Coche(String matrícula, int puertas) {
        super(matrícula);
```

Llamada al constructor de la superclase.

```
        numPuertas = puertas;
    }

    public int puertas(){
        return numPuertas;
    }
}

public class Camion extends Vehiculo{
    private Remolque remolque;

    public Camion(String matrícula) {
        super(matrícula); <-------------------- Llamada al constructor de la superclase.
        remolque = null;
    }

    public void ponRemolque(Remolque rem){
        remolque = rem;
    }

    public void quitaRemolque(){
        remolque = null;
    }
                                     Permitido porque acelerar() en Vehículo indica que se pueden lanzar estas excepciones.
    public void acelerar(double cantidad)throws DemasiadoRapidoException{
        if (remolque != null  && getVelocidad() + cantidad > 100) <---- Método heredado.
            throw new DemasiadoRapidoException();
        super.acelerar(cantidad);
    }
     public String toString(){
         if (remolque != null){
            return super.toString()+". Lleva un " + remolque.toString();
         }else{                ^---- Llamada a método toString() de la superclase.
            return super.toString();
         }
     }
}

public class Remolque {
    private int peso;

    public Remolque(int peso) {
        this.peso = peso;
    }

    public String toString(){
        return "Remolque con un peso " + peso;
    }
}

public class DemasiadoRapidoException extends Exception {
```

```
        public DemasiadoRapidoException() {
        }

        public DemasiadoRapidoException(String msg) {
            super(msg);
        }
}
```

Comentario: En el método acelerar() de Camión, para obtener la velocidad se llama al método getVelocidad(). Aunque los objetos de tipo Camión tienen un atributo velocidad, no pueden acceder a él directamente ya que está declarado como privado en la superclase. Para acceder a él utilizan el método (también heredado) getVelocidad(). Si se quisiese acceder directamente a él, el atributo tendría que ser declarado protected. Esto aumentaría el acoplamiento entre las clases, algo que es poco recomendable.

Ejercicio 5.19:

Utilizando las clases del ejercicio anterior, implemente una aplicación que haga lo siguiente:

a) Declare y cree un array de 4 vehículos.
b) Cree 2 camiones y 2 coches y los añada al array.
c) Suponiendo que no se sabe en qué celdas están los coches y en cuáles los camiones:
   1. Ponga un remolque de 5000 Kg a los camiones del array.
   2. Acelere todos los vehículos.
   3. Escriba por pantalla la información de todos los vehículos.

Planteamiento: Para poder almacenar en un array de objetos de tipo Coche y Camión, hay que declarar el array de tipo Vehículo, ya que Coche y Camión heredan de ella. Y por las propiedades de la herencia, algo declarado del tipo de la superclase puede referenciar a un objeto creado como instancia de una subclase.

Suponiendo que no se conoce la posición de los Camiones y los Coches (como pide el ejercicio) se debe recorrer el array preguntando a cada objeto si es de tipo Camión. Para los objetos que sean camiones hay que hacer una conversión a la subclase Camión para poder llamar al método ponRemolque(), ya que este método no pertenece a la clase Vehículo.

Para acelerarlos todos se llama al método acelerar() que está implementado en la clase Vehículo, es heredado por Coche y sobreescrito por Camión. Cuando se llama a acelerar() con una referencia a Camión se ejecuta el método acelerar() de Camión por ligadura dinámica. Hay que capturar las excepciones que se puedan lanzar con una sentencia try-catch.

Para escribir la información por pantalla se llama al método toString(). Otra vez es la ligadura dinámica la que se encarga de llamar al método toString() implementado en Camión cuando se esté referenciando a un Camión y al método toString() implementado en Vehículo cuando se referencie a un Coche. Este método se hereda, ya que no se ha sobreescrito en Coche.

Solución:

```
public class Main {
    public static void main(String[] args) {
        Vehiculo[] vehiculos = new Vehiculo[4];
        vehiculos[0] = new Coche("1324ABC",3);
        vehiculos[1] = new Camion("1111ABC");
        vehiculos[2] = new Coche("1333ABC",3);
        vehiculos[3] = new Camion("2222ABC");
        for (Vehiculo v: vehiculos){
            if (v instanceof Camion){
                ((Camion)v).ponRemolque(new Remolque(5000));
```

Comprueba que es de tipo Camion.

Convierte a Camion para poder llamar a ponRemolque().

```
                }
            }
            for (Vehiculo v: vehículos){
                try{
                    v.acelerar(80); ←
                }catch(DemasiadoRapidoException e){
                    System.out.println("Error: algún vehículo se ha acelerado más de lo debido");
                }
            }

            for (Vehiculo v: vehículos){
                System.out.println(v); ←
            }
        }
    }
```

En la cabecera de acelerar() se indica que puede lanzar DemasiadoRapidoException.

Se llama automáticamente al método toString().

Nota: El `try-catch` va dentro del `for`, si estuviese al revés, cuando se lanzase una excepción, no se seguirían acelerando el resto de los vehículos.

HERENCIA FORZADA Y CLASES ABSTRACTAS

Ejercicio 5.20:

Una pila y una cola son estructuras muy similares. En ambas se añaden elementos por uno de los extremos y se extraen elementos por uno de los extremos también. La diferencia es que en la pila se añaden y se extraen por el mismo extremo y en la cola por extremos distintos.

Aprovechando esto, escriba una superclase de ambas que implemente la parte común que se llame ColeccionSimple. Los métodos son los siguientes:

- *estaVacia(): devuelve true si la colección está vacía y false en caso contrario.*
- *extraer(): devuelve y elimina el primer elemento de la colección.*
- *primero(): devuelva el primer elemento de la colección.*
- *añadir(): añade un objeto por el extremo que corresponda.*
- *toString(): devuelve en forma de String la información de la colección.*

Posteriormente escriba las clases Pila y Cola que hereden de ColeccionSimple implementando la parte que las diferencia. Esta ColeccionSimple debe almacenar objetos de cualquier tipo, es decir objetos de tipo Object.

Planteamiento: Para implementar la clase `ColeccionSimple` se va a utilizar una `LinkedList` de `Object`. En Java todas las clases heredan de `Object`, por lo que permite almacenar objetos de cualquier tipo (los tipos primitivos se envolverán en sus clases correspondientes). La relación será de composición.

En esta clase se indicará que contiene un método `añadir()`, pero no se implementará, ya que es diferente para las dos subclases. Este método es abstracto. Es donde se fuerza a la herencia.

El resto de los métodos se implementa mediante una llamada al método correspondiente de `LinkedList`, obteniendo los elementos por el principio de la lista.

La clase `Pila` hereda de `ColeccionSimple` y sobreescribe el método `añadir()`. Para esta clase, `añadir()` consiste en añadir por el principio de la lista.

La clase `Cola` hereda de `ColeccionSimple` y sobreescribe el método `añadir()`. Para esta clase, `añadir()` consiste en añadir por el final de la lista.

El atributo lista es privado en `ColeccionSimple` y las subclases necesitan acceder a él para añadir elementos. Para solucionar este problema, se implementa en la clase `ColeccionSimple` un método `getLista()` que sea `protected`.

Solución:

```
import java.util.LinkedList;

public abstract class ColeccionSimple {
    private LinkedList<Object> lista;

    public ColeccionSimple() {
        lista = new LinkedList<Object>();
    }

    public abstract void añadir(Object o);

    public Object extraer(){
        return lista.removeFirst();
    }

    public boolean estaVacia(){
        return lista.isEmpty();
    }

    public Object primero(){
        return lista.getFirst();
    }

    public String toString(){
        return lista.toString();
    }

    protected LinkedList<Object> getLista(){
        return lista;
    }
}

public class Pila extends ColeccionSimple{

    public Pila() {
    }

    public void añadir(Object o){
        getLista().addFirst(o);
    }
}

public class Cola extends ColeccionSimple{

    public Cola() {
    }
    public void añadir(Object o){
        getLista().addLast(o);
    }
}
```

Declaración de clase abstracta.

LinkedList declarado para que almacene Object.

Método abstracto, especificado pero no implementado.

Método protected para que puedan acceder a él las subclases.

Java entiende que se llama al constructor de la superclase por defecto: super().

Se sobreescribe el método añadir(), que estaba sin implementar.

Java entiende que se llama al constructor de la superclase por defecto: super().

Se sobreescribe el método añadir(), que estaba sin implementar.

Nota: La clase `LinkedList` es genérica. Para el ejercicio se ha declara que almacene objetos de tipo `Object`. Para más información acerca de colecciones consulte el Capítulo 6. Para más información acerca de genericidad consulte el Capítulo 9.

Comentario: La clase `ColeccionSimple` y sus descendientes se han implementado para almacenar objetos de tipo `Object`. Es más correcto declarar estas clases genéricas con tipo parametrizado, de manera que el cliente elija qué tipo de datos quiere almacenar. Para más información acerca de cómo hacer la clase `ColeccionSimple` genérica consulte el Ejercicio 9.6 del Capítulo 9.

Ejercicio 5.21:

Escriba una aplicación en la que se implementen dos métodos:

a) rellenar(): recibe por parámetro un objeto de tipo ColeccionSimple y añade los números del uno al diez.
b) imprimirYVaciar(): recibe por parámetro un objeto de tipo ColeccionSimple y va extrayendo e imprimiendo los datos de la colección hasta que se quede vacía.
c) En la aplicación principal:
   1. Crear un objeto de tipo Pila, llamar a los métodos rellenar() e imprimirYVaciar() pasando este objeto por parámetro.
   2. Crear un objeto de tipo Cola, llamar a los métodos rellenar() e imprimirYVaciar() pasando este objeto por parámetro.
   3. Por último añadir a la pila varios objetos de distinto tipo. Llamar a imprimirYVaciar pasando este objeto por parámetro.

Planteamiento:

a) `rellenar()`: se realiza un `for` con un `int` que vaya de uno a diez y se llama a `añadir()` pasando este entero por parámetro. Java se encarga de convertir el `int` en `Integer`.
b) `imprimirYVaciar()`: se realiza un `while` en el que mientras no esté vacía se imprime a pantalla el resultado del método `extraer()`.
c) En la aplicación principal:
   1. Se crea el objeto de tipo `Pila`. Al pasarlo por parámetro a los métodos que reciben `ColeccionSimple` no existe ningún problema. Como `Pila` hereda de `ColeccionSimple`, la conversión de tipo hacia arriba se realiza automáticamente. `imprimirYVaciar()` imprimirá en el contrario al que se han añadido.
   2. Con la conversión hacia arriba de `Cola` a `ColeccionSimple` sucede lo mismo que en el caso anterior. `imprimirYVaciar()` en el orden en el que se han añadido.
   3. Se llama al método `añadir()` pasando objetos de distinto tipo, como recibe `Object`, los aceptará todos. Si se añade alguno de tipo primitivo, Java se encarga de envolverlo en un objeto de su clase correspondiente.

Solución:

```java
public class Main {

    public static void rellenar(ColeccionSimple c){
        for (int i=1; i<=10; i++){
            c.añadir(i);        // Java se encarga de convertir el int a Integer.
        }
    }

    public static void imprimirYVaciar(ColeccionSimple c){
        while(!c.estaVacia()){
            System.out.println(c.extraer());   // Se llama automáticamente al método toString() del objeto que devuelve.
```

```
        }
    }

    public static void main(String[] args) {
        Pila p = new Pila();
        rellenar(p);
        System.out.println("Datos de la pila");
        imprimirYVaciar(p);

        Cola c = new Cola();
        rellenar(c);
        System.out.println("Datos de la cola");
        imprimirYVaciar(c);

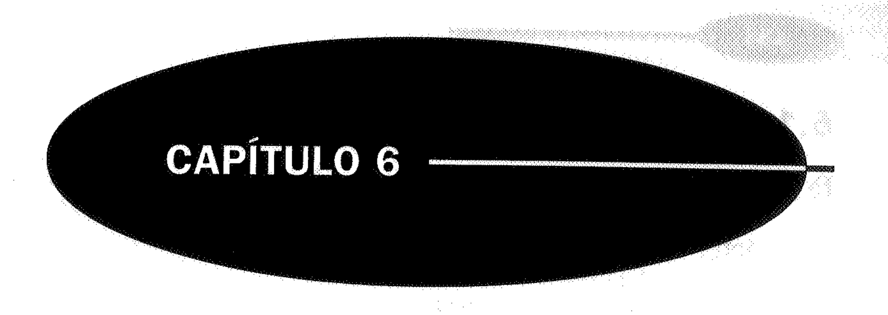
        p.añadir("Primero");
        p.añadir(new Boolean(true));
        p.añadir(7.5);
        p.añadir(c);
        System.out.println("Datos de la pila con objetos de distinto tipo");
        imprimirYVaciar(p);
    }
}
```

p se convierte a ColeccionSimple automáticamente.

c se convierte a ColeccionSimple automáticamente.

Al añadir objetos de distinto tipo se convierten de forma apropiada.

Nota: En Java todas las clases heredan directa o indirectamente de `Object`, por lo que una referencia declarada de tipo `Object`, puede referenciar a un objeto creado de cualquier tipo.



Estructuras de almacenamiento

6.1 ARRAYS

6.1.1 Declaración

En Java un array se declara de las siguientes formas:

```
int[] nombreArray1;
int nombreArray2[];
```

Ambas declaraciones son equivalentes. La primera línea declara un array de enteros de nombre nombreArray1. Es decir, nombreArray1 es una referencia a una estructura donde se pueden guardar muchos valores del tipo int. La segunda línea declara que cada elemento de la forma nombreArray2[] es del tipo int.

Se denomina tipo base del array al tipo de los elementos del array. No existe ninguna restricción al tipo base de un array: puede ser un tipo primitivo (int, float, etc.), un enumerado o cualquier clase (String, Punto, etc.).

6.1.2 Creación

Los arrays son objetos, por lo que para poder utilizarlos se necesita en primer lugar crear el objeto. Como para cualquier otro objeto el valor inicial por defecto de un array es null. La creación de los arrays se hace de la siguiente forma:

```
nombreArray1 = new int[20];
nombreArray2 = new int[100];
```

A partir de entonces, el array nombreArray1 permite guardar 20 números enteros y el array nombreArray2 permite guardar 100 números enteros. Al crear el array, cada uno de sus elementos se inicializa al valor por defecto, es decir, 0 para los números, false para los boolean, \u0000 para los caracteres y null para las referencias a objetos.

Los arrays también se pueden crear cuando se declaran.

```
int[] nombreArray1   = new int[20];
int   nombreArray2[] = new int[100];
```

6.1.3 Inicialización estática

Es posible inicializar los elementos del array a la vez que se crean de la siguiente forma:

```
int[] arrayEnteros   = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
String[] díasSemana = {"Lunes", "Martes",
                       "Miércoles", "Jueves",
                       "Viernes", "Sábado", "Domingo"};
```

En el primer caso, `arrayEnteros` se ha declarado como un array de 10 números enteros, con valor inicial del 1 al 10. En el segundo caso, el array `díasSemana` se ha creado con siete objetos de la clase `String`, cuyo valor inicial son los nombres de los días de la semana.

⇨ Sobre la inicialización estática consulte los Ejercicios 6.1, 6.13 y 6.15.

6.1.4 Acceso a los valores

Cuando se desea acceder a los valores de un array de tamaño N, el primer elemento se encuentra en la posición 0 y el último se encuentra en la posición N – 1. Cualquier intento de hacer referencia a un elemento que no esté entre 0 y N – 1 genera la excepción `ArrayIndexOutOfBoundsException`.

De esta forma se puede escribir:

```
System.out.println("El quinto día de la semana es: " + díasSemana[4]);
```

6.1.5 Tamaño de un array

Para conocer el número de elementos de un array se utiliza su atributo `length`, de la siguiente forma:

```
System.out.println("El array díasSemana tiene "  + díasSemana.length + " elementos.");
```

6.2 ARRAYS MULTIDIMENSIONALES

6.2.1 Declaración y creación

Un array declarado de la siguiente forma representa una tabla de dos dimensiones:

```
int[][] tabla;
```

Para crear un array de dos dimensiones hay que indicar los tamaños de las dos dimensiones. Por ejemplo, para declarar una tabla de 4 por 7 elementos se utilizaría la sentencia de Java:

```
int[][] tabla = new int[4][7];
```

Los arrays pueden tener más de dos dimensiones. Para declarar un array de tres dimensiones se haría:

```
int[][][] cubo = new int[4][7][6];
```

⇨ Sobre la creación de arrays multidimensionales consulte los Ejercicios 6.18 al 6.24.

6.2.2 Acceso

Para acceder a cualquier variable del array de dos dimensiones hay que indicar en qué posición de cada una de las dimensiones se encuentra la variable a la que se desea acceder.

```
tabla[1][3] = 55
```

6.2.3 Tamaño de un array

El atributo `length` permite conocer el tamaña de un array. Si un array tiene más de una dimensión cada dimensión, tiene su propio tamaño. Se puede utilizar el atributo `length` de la siguiente forma:

```
tamaño  = tabla.length; //tamaño de la primera dimension, es decir 4
tamaño2 = tabla[0].length; //tamaño de la primera fila, es decir 7
```

➪ Sobre el manejo de las dimensiones de una tabla consulte los Ejercicios 6.18, 6.19, 6.23 y 6.24.

6.2.4 Array de dos dimensiones no rectangulares

La tabla declarada en la sección anterior tenía un tamaño de 4x7, es decir, 28 elementos. Esta tabla está compuesta por 4 arrays, donde cada uno de ellos es un array de 7 variables de tipo `int`. Sin embargo, cada uno de los 4 arrays podría haberse creado con arrays de distinto tamaño:

```
int[][] tabla = new int[4][];
tabla[0] = new int[3];
tabla[1] = new int[5];
tabla[2] = new int[7];
tabla[3] = new int[4];
```

➪ Sobre la creación de arrays multidimensionales no rectangulares consulte los Ejercicios 6.17 y 6.23

6.3 API DE MANEJO DE ARRAYS

6.3.1 Uso con arrays de tipos primitivos

La API del lenguaje proporciona la clase ya definida `Arrays` para realizar múltiples operaciones con arrays. Algunas de las más interesantes son rellenar el array, buscar si un dato está en el array y ordenar los datos de un array.

Suponga que se ha creado un array de datos de tipo int de la siguiente forma:

```
int[] a = new int[20];
```

Para rellenar todo el array de enteros con el número 23 utilice:

```
Arrays.fill(a, 23);
```

Para rellenar la parte del array entre las posiciones 4 y 10 (ambas inclusive) con el número 23 utilice:

```
Arrays.fill(a, 4, 11, 23); //La última posición indicada no se rellena.
```

Para ordenar el array en orden ascendente utilice:

```
Arrays.sort(a);
```

Para ordenar sólo una parte del array entre las posiciones 6 y 15 (ambas inclusive) utilice:

```
Arrays.sort(a, 6, 16);
```

Para buscar un dato en un array, éste debe estar previamente ordenado. Para buscar el elemento 23 en un array de enteros ordenado utilice.

```
int posición = Arrays.binarySearch(a, 23);
```

Si posición es un número positivo, en esa posición hay un entero con el valor buscado. Si hay más de un entero con el valor que se busca, no se indica nada sobre cuál de ellos se devuelve. Si el número buscado no se encuentra en el array, se devuelve un número negativo, cuyo valor en positivo es el lugar esperado para el valor buscado.

6.3.2 Uso con arrays de objetos

Las funciones anteriores también se pueden utilizar con arrays de objetos, en particular el método `Arrays.fill()` rellena un array, total o parcialmente, con referencias a un objeto dado. En el caso de utilizar el método `Arrays.sort()` la clase debe implementar la interfaz `Comparable`, como en el siguiente ejemplo que implementa una clase `Alumno` que tiene un atributo número. Los alumnos se ordenarán de forma ascendente por su atributo número. La clase quedará entonces:

```
public class Alumno implements Comparable<Alumno>, Comparator<Alumno>{
    private int numero;
    // El resto de los atributos y métodos de la clase

    public int compareTo (Alumno alum){
        if (numero > alum.getNumero())
            return 1;
        if (numero < alum.getNumero())
            return -1;
        return 0;
    }

    public int compare (Alumno a1, Alumno a2){
        return a1.getApellidos().compararеTo(a2.getApellidos());
    }
}
```

En este caso, para ordenar un array de alumnos se utilizaría:

```
Alumno[] alumnos = new Alumno[40];
// Se rellena el array con los alumnos
Arrays.sort(alumnos);
```

Si en un momento posterior quisiera ordenar los alumnos por otro criterio se puede utilizar un objeto comparador de alumnos. En el ejemplo anterior se ha implementado también la interfaz `Comparator` para ordenar los alumnos por apellidos. Para ello se utilizaría la sentencia:

```
// Se ordenan los alumnos por apellidos
Arrays.sort(alumnos, alumnos[0]);
```

⇨ Sobre el uso de los métodos de la API de la clase Arrays consulte los Ejercicios 6.17 y 6.25 a 6.28.

6.4 COLECCIONES

En muchos programas es necesario guardar gran cantidad de datos de distintos tipos de objetos. Java dispone de todo un conjunto de clases predefinidas que implementan estructuras de control muy potentes para el manejo de información. En esta sección se presentan algunas de ellas para que pueda ver cómo funcionan. Para el resto puede consultar la API del lenguaje en la documentación.

En la Tabla 6.1 se puede ver resumida la estructura de las colecciones. En ella puede ver las principales interfaces y las clases más útiles que heredan de ellas. En particular resulta interesante destacar las propiedades de la interfaz `Collection`, ya que las implementaciones más utilizadas derivan de ella.

Tabla 6.1. Estructura de las principales clases de colecciones.

		Implementaciones				
		Tabla Hash	Array redimensionable	Arbol balanceado	Listas enlazadas	Tabla Hash + Listas enlazadas
Interfaces	Set	HashSet		TreeSet		LinkedHashSet
	List		ArrayList		LinkedList	
	Map	HashMap		TreeMap		LinkedHashMap

Una propiedad muy interesante es que en muchas situaciones se puede utilizar cualquiera de las clases que aparecen citadas en la Tabla 6.1 de forma independiente de a qué clase pertenezca, ya que todas ellas implementan la interfaz `Collection`.

Por ejemplo, todos los objetos que se hayan añadido a una cualquiera de las clases de la Tabla 6.1, se pueden recorrer utilizando un iterador. Suponga que se ha declarado un objeto llamado `colección` de cualquiera de las clases anteriores y se le han añadido objetos. Para imprimir por pantalla todos los datos del objeto `colección` se puede realizar de la siguiente forma:

```
public void imprimir(){
    Iterator iterador = colección.iterator();
    while (iterador.hasNext())
        System.out.println(iterador.next());
}
```

Un iterador tiene dos métodos básicos: `hasNext()`, que indica si quedan todavía más elementos y `next()`, que devuelve el siguiente elemento. Con estos dos métodos resulta muy sencillo recorrer todos los datos.

De todas formas para esta tarea resulta muy sencillo también utilizar directamente un bucle `for` que vaya tomando los valores de los elementos de la colección de la siguiente forma (siendo `Elemento` la clase de la que se han definido los elementos que se añaden al objeto `coleccion`):

```
public void imprimir(){
    for(Object elem : colección){
        System.out.println(elem);
    }
}
```

En la Tabla 6.2 se describen los principales métodos que se pueden utilizar en cualquiera de las clases que implementan la interfaz Collection.

Tabla 6.2. Principales métodos de la interfaz Collection.

Métodos	Funcionalidad
int size()	Devuelve el número de objetos en la colección.
boolean add(E elemento)	Añade el elemento de la clase E a la colección. Devuelve true si la colección ha cambiado tras la operación.
boolean remove(Object elemento)	Elimina el elemento dado que case con uno de la colección. Devuelve true si la colección ha cambiado tras la operación.
Iterator<E> iterator()	Devuelve un iterador para los elementos de esta colección.
void clear()	Elimina todos los elementos de la colección.
boolean contains(Object elemento)	Devuelve true si la colección contiene un elemento que case con el dado.
Object[] toArray()	Devuelve un array con todos los elementos de la colección.

6.4.1 ArrayList y LinkedList

Estas dos clases funcionan como listas de objetos. ArrayList es muy utilizada, pues se utiliza de forma similar a un array. LinkedList funciona como una lista enlazada y tiene métodos en este sentido.

La diferencia entre ellas es la siguiente:

- En ArrayList aunque funciona como un array, la diferencia es que no hay que preocuparse por el tamaño. El tamaño del almacenamiento crece él sólo cuando se van añadiendo elementos. Es más eficiente que LinkedList en el manejo de los datos.
- LinkedList permite manejar listas de objetos y manejarlos como una cola o una pila.

Para probar cómo crece un ArrayList, se puede comprobar con el siguiente fragmento de código, donde se crea un ArrayList para 10 objetos de la clase String:

```
ArrayList<String> listaString = new ArrayList<String>(10);
for(int i = 0; i < 20; i++){
    listaString.add("Cadena número " + i);
}
System.out.println("La colección tiene " + listaString.size() + " elementos");
```

Como puede imaginar, este fragmento imprime correctamente que la colección tiene 20 elementos.

En la Tabla 6.3 se describen los principales métodos de ArrayList, además de los ya indicados en Collection.

En la Tabla 6.4 se describen los principales métodos de LinkedList, además de los ya indicados en Collection.

⇨ Sobre el uso de las clases ArrayList y LinkedList consulte los Ejercicios 6.25 al 6.28.

6.4.2 HashSet y TreeSet

Las dos clases HashSet y TreeSet implementan la interfaz Set. Eso hace que estas dos colecciones sólo guarden objetos que no estén repetidos. Si se intenta añadir un elemento que ya está, no lo añade devolviendo false. De hecho, esto hace que funcionen como un conjunto.

Tabla 6.3. Principales métodos de la clase `ArrayList`.

Métodos	Funcionalidad
`public void add(int pos, E elemento)`	Añade un elemento en la posición indicada. Los elementos que hubiese se desplazan hacia la derecha.
`public E remove(int pos)`	Elimina el elemento de la posición indicada. El resto de elementos se desplaza hacia la izquierda. Devuelve el elemento eliminado.
`public E get(int pos)`	Devuelve el elemento de la posición indicada.
`public E set(int pos, E elemento)`	Sustituye el elemento de la posición indicada por el nuevo elemento devolviendo el que había antes.
`public int indexOf(Object elemento)`	Devuelve la posición donde se encuentra la primera ocurrencia del elemento indicado, o -1 si no se encuentra. Los elementos se comparan utilizando el método `equals()`.
`public int lastIndexOf(Object elemento)`	Igual que `indexOf()` pero devuelve la última ocurrencia del objeto.

Tabla 6.4. Principales métodos de la clase `LinkedList`.

Métodos	Funcionalidad
`void addFirst(E elemento)` `void addLast(E elemento)`	Añade el elemento al principio y al final de la lista, respectivamente.
`Public E getFirst()` `public E getLast()`	Devuelve el primero y el último elemento de la lista, respectivamente.
`public void removeFirst()` `public void removeLast()`	Elimina el primero y el último elemento de la lista, respectivamente.

`public void add(int pos, E elemento)`
`public E remove(int pos)`
`public E get(int pos)`
`public E set(int pos, E elemento)`
`public int indexOf(Object elemento)`
`public int lastIndexOf(Object elemento)`
	Iguales que para `ArrayList`.

Por ejemplo, el siguiente código crea un objeto `TreeSet` y le añade 20 cadenas, pero varias repetidas. Al final se imprime cuantas cadenas se tienen en la colección. Como era de esperar escribe en pantalla que hay sólo 5 elementos.

```
TreeSet<String> arbolString = new TreeSet<String>();
for(int i = 0; i < 20; i++){
   arbolString.add("Cadena número " + i%5);
}
System.out.println("La colección tiene " + arbolString.size() + " elementos");
```

La diferencia entre `HashSet` y `TreeSet` es la siguiente:

- HashSet realiza todas las operaciones habituales: añadir un elemento, borrarlo o comprobar si está en tiempo constante, independiente del número de elementos.

  Si contiene muchos elementos resulta muy costoso recorrerlos mediante un `Iterator`. No se garantiza el orden. De hecho, el orden de los elementos puede cambiar si se añaden más y tiene que crecer el contenedor.

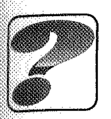
- TreeSet realiza todas las operaciones anteriores en tiempo logarítmico con el número de elementos. Además mantiene todos los elementos ordenados en su orden natural o de acuerdo a como indique el Comparator que se indique en el constructor.

Los principales métodos de HashSet son los ya indicados para la interfaz Collection.
En la Tabla 6.5 se describen los principales métodos de TreeSet, además de los ya indicados en Collection.

Tabla 6.5. Principales métodos de la clase TreeSet.

Métodos	Funcionalidad
public E first() public E last()	Devuelve el menor y el mayor, respectivamente, de los elementos del conjunto ordenado.
SortedSet<E> headSet(E elemento) SortedSet<E> tailSet(E elemento)	Devuelve el conjunto de elementos que son menores que, o mayores o iguales que, respectivamente, el elemento del parámetro.

⇨ Sobre el uso de la clase TreeSet consulte los Ejercicios 6.25 al 6.28.



Problemas resueltos

ARRAYS

Ejercicio 6.1:

Escriba una clase, de nombre EjArrays, cuyo método main() implemente un programa que lleve a cabo las siguientes acciones:

a) Declarar y construir un array de enteros, de nombre arrDig1, que almacene los números del 0 al 9 en orden creciente.
b) Declarar y construir un array de enteros, de nombre arrDig2, que almacene los números del 9 al 0.
c) Declarar y construir un array de caracteres, de nombre arrCh1, que almacene las 5 vocales en minúscula.
d) Declarar y construir un array de caracteres, de nombre arrCh2, que almacene las 5 primeras letras del abecedario en minúscula.
e) Concatenar las vocales del array arrCh1 a una cadena de caracteres de nombre cadCh1.
f) Obtener una cadena de caracteres, de nombre cadCh2, con las 5 letras del array arrCh2.
g) Mostrar por pantalla las cadenas con las 5 vocales y las 5 primeras letras del abecedario.
h) Mostrar por pantalla un texto donde se intercalan las vocales con las 5 primeras letras del abecedario.
i) Mostrar por pantalla el resultado de sumar cada posición del array arrDig1 con su correspondiente en el array arrDig2

Planteamiento:

a) Se declara el array `arrDig1` y se inicializa de forma estática con sus elementos especificando directamente los números del 0 al 9.
b) Se declara y se construye el array `arrDig2` como un array de enteros de 10 variables. Se recorren las posiciones mediante un bucle para asignar los números del 9 al 0. Las posiciones del array se recorren en orden ascendente, por lo que es necesario calcular apropiadamente el valor asignado en cada iteración del bucle. Se aprovecha el atributo `length` del array y la variable índice para conseguirlo.
c) Al igual que `arrDig1`, el array `arrCh1` se declara estáticamente con sus elementos, en este caso las 5 vocales en minúsculas.
d) El array `arrCh2` se declara y construye como un array de 5 variables de tipo `char`. Para asignar las 5 primeras letras del abecedario a sus posiciones se utiliza un bucle que las recorra secuencialmente y se aprovecha la disposición consecutiva de las letras en el código Unicode para ir obteniendo las siguientes letras. Internamente, un carácter se almacena como el valor entero correspondiente a su código Unicode. Por ello, incrementar en una unidad este valor equivale a pasar al siguiente carácter del citado código.
e) Para obtener una cadena con las 5 vocales se declara un objeto `String` al que se van concatenando dentro de un bucle las vocales del array `arrDig1`.
f) La cadena de caracteres con las 5 primeras letras del abecedario se obtiene aprovechando un constructor de la clase `String` que recibe por parámetro un array de caracteres. Para la solución se emplea el array `arrCh2`.
g) Para mostrar las dos cadenas, basta con imprimirlas directamente con el método `System.out.println()`.
h) Para mostrar las primeras letras del abecedario intercaladas con las 5 vocales, se utiliza un bucle para ir obteniendo el carácter en la misma posición de cada cadena. Estos caracteres se irán concatenando a una nueva cadena que finalmente se mostrará por pantalla.
i) Para sumar las posiciones de los arrays de enteros, se utiliza un bucle para recorrerlas y que, en cada iteración, muestre la suma de dos de sus posiciones.

Solución:

```
public class EjArrays {
```

```
    public static void main(String[] args) {
        int[] arrDig1 = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
        int[] arrDig2;
        arrDig2 = new int[10];
        for(int i = 0; i < arrDig2.length; i++){
            arrDig2[i] = arrDig2.length-1-i;
        }
        char[] arrCh1 = {'a', 'e', 'i', 'o', 'u'};
        char[] arrCh2;
        arrCh2 = new char[5];
        for(int i = 0; i < arrCh2.length; i++){
            arrCh2[i] = (char)("a"+i);
        }
        String cadCh1 = "";
        for(int i = 0; i < arrCh1.length; i++){
            cadCh1 += arrCh1[i];
        }

        String cadCh2 = new String (arrCh2);
        System.out.println("Cadena 5 vocales: " + cadCh1);
        System.out.println("Cadena 5 primeras letras: " + cadCh2);

        String intercaladas = "";
        for(int i = 0; i < cadCh1.length(); i++){
            intercaladas += cadCh1.charAt(i);
            intercaladas += cadCh2.charAt(i);
        }

        System.out.println("Vocales y letras: " + intercaladas);
        for(int i = 0; i < arrDig1.length && i < arrDig2.length ;i++){
            System.out.println("La suma de " + arrDig1[i] +
                               " mas " + arrDig2[i] +
                               " es: " + (arrDig1[i] + arrDig2[i]) );
        }
    }
}
```

Se declaran e inicializan de forma estática todos sus elementos.

El valor del atributo `length` sirve para obtener el valor asignado a la siguiente posición.

Se concatenan los elementos del array a la cadena.

Se usa un constructor de la clase `String`.

Se concatena alternativamente un carácter de cada cadena.

Los paréntesis son necesarios para obtener suma y no concatenación.

Ejercicio 6.2:

Escriba un método, de nombre mostrarArrayPantalla1, que reciba un array de enteros por parámetro y muestre sus valores por pantalla, cada uno en una línea.

Planteamiento: Como hay que mostrar todos los valores del array, se necesita acceder a todas sus posiciones. Para ello se utiliza una variable que actuará como índice. Su valor se irá incrementando con cada repetición del bucle. También se puede utilizar un `for` que recorra el array completo.

Parámetros: El método recibe por parámetro un array con los valores enteros que se mostrarán por pantalla.

Valor de retorno: El método no necesita devolver ningún valor de retorno por lo que se utiliza la cláusula `void`.

Solución 1:

```
public void mostrarArrayPantalla1(int[] arr){
```

```
        for (int i = 0; i < arr.length; i++){
            System.out.println(arr[i]);
        }
    }
```

Solución 2:

```
public void mostrarArrayPantalla1(int[] arr){
    for (int v : arr){
        System.out.println(v);
    }
}
```

Aviso: Aquellos métodos que reciban por parámetro referencias a arrays deberían comprobar que no son nulas (tienen valor `null`) si se quiere garantizar un adecuado comportamiento en tiempo de ejecución.

Ejercicio 6.3:

Escriba un método, de nombre mostrarArrayPantalla2, que reciba por parámetro un array de enteros y muestre sus valores por pantalla separados por comas.

Planteamiento: Se recorren todas las posiciones del array para mostrar su valor por pantalla, escribiendo el valor y una coma. Para evitar que el último elemento aparezca seguido de una coma, se escriben dentro de un bucle todos los elementos del array excepto el último. Finalmente, se muestra el último elemento del array.

Parámetros: El método recibe por parámetro un array con los valores enteros que se mostrarán por pantalla.

Valor de retorno: El método no necesita devolver ningún valor de retorno por lo que se utiliza la cláusula `void`.

Solución:

```
public void mostrarArrayPantalla2(int[] arr){
    for (int i = 0; i < arr.length-1; i++){
        System.out.print(arr[i]+ ", ");
    }
    System.out.println(arr[arr.length-1]);
}
```

Se muestra el último elemento del array. No se puede utilizar el valor de i, porque su ámbito de validez se restringe al bucle `for`.

Ejercicio 6.4:

Escriba un método, de nombre obtenerArrayComoString, que reciba un array de enteros por parámetro y devuelva una cadena de caracteres con su contenido.

Planteamiento: Se recorren todas las posiciones del array para ir concatenando su contenido a una cadena de caracteres. Para ello, en primer lugar, se declara y construye un objeto de la clase `String`. El objeto es inicializado con la cadena vacía y dentro de un bucle se le irán concatenando los valores del array.

Parámetros: El método recibe por parámetro un array con los valores enteros.

Valor de retorno: El método devolverá una cadena de caracteres compuesta por los elementos contenidos en el array, es decir, un objeto de la clase `String`.

Solución:

```
public String obtenerArrayComoString(int[] arr){
```

```
    String resul = "";
    for (int i : arr){
        resul += i;
    }
    return resul;
}
```

Para concatenar valores a la cadena, es necesario inicializarla a la cadena vacía.

Comentario: Hay que tener en cuenta que a los objetos de la clase `String` no se les puede asignar directamente valores que no sean cadenas. Por otro lado, no se puede modificar el contenido de un objeto de la clase `String`. Hay que tener en cuenta que cualquier operación con un objeto de la clase `String` crea un nuevo objeto de dicha clase. Por eso, se utiliza el recurso de la concatenación, que permite obtener la representación textual de valores de tipos primitivos.

Ejercicio 6.5:

Escriba un método, de nombre completarArray1, que reciba un array de enteros por parámetro y lo rellene de forma que asigne a cada posición el valor de su índice.

Planteamiento: Para asignar valores al array de la forma solicitada, habrá que recorrer todas sus posiciones mediante un bucle y asignarles el valor correspondiente a su índice. Para ello se aprovecha la variable utilizada como índice para asignarla a la posición que indexa. Dado que el número de iteraciones a realizar se corresponderá con la longitud del array y, por tanto, se encuentra acotado de antemano, se recomienda la utilización de un bucle de tipo `for`.

Parámetros: El método recibe por parámetro un array de números enteros.

Valor de retorno: El método no necesita devolver ningún valor de retorno por lo que se utiliza la cláusula `void`.

Solución:

```
public void completarArray1(int[] arr){
    for (int i = 0; i < arr.length; i++){
        arr[i] = i;
    }
}
```

A cada posición del array se le asigna el valor de su índice.

Comentario: Cuando un array primitivo se recorre para modificar sus posiciones no se puede utilizar el tipo de bucle `for` del ejercicio anterior.

Ejercicio 6.6:

Escriba un método, de nombre completarArray2, que reciba un array de enteros por parámetro y lo rellene de forma que asigne a todas sus posiciones un valor que también se recibirá por parámetro.

Planteamiento: Se recorre el array con un bucle `for` y se le asigna a todas las posición del array el valor especificado por parámetro.

Parámetros: El método recibe por parámetro un array de números enteros y el valor que se asignará a todas sus posiciones.

Valor de retorno: El método no necesita devolver ningún valor de retorno por lo que se utiliza la cláusula `void`.

Solución:

```
public void completarArray2(int[] arr, int valor){
```

```
    for (int i = 0; i < arr.length; i++){
        arr[i] = valor;   <------   A cada posición del array se le
    }                               asigna el valor recibido.
}
```

Comentario: Recuerde que esta misma función la cumple el método `Arrays.fill()`.

Ejercicio 6.7:

Escriba un método, de nombre completarArray3, que reciba un array de enteros por parámetro y lo rellene de forma que contenga tantos números pares, a partir del cero, como permita su capacidad.

Planteamiento: Para asignar valores al array de la forma solicitada, se utilizará un bucle que, mientras recorre las posiciones del array, vaya calculando también el número par que se asignará a cada posición.

Parámetros: El método recibe por parámetro un array de números enteros.

Valor de retorno: El método no necesita devolver ningún valor de retorno por lo que se utiliza la cláusula `void`.

Solución:

```
public void completarArray3(int[] arr){
    for (int i = 0, numPar = 0; i < arr.length; i++, numPar += 2){
        arr[i] = numPar;
    }
}
```

Aviso: El tipo `int` permite un rango de valores entre -2^{31} y 2^{31}-1, por lo que el mayor número par, como tipo `int`, que se puede almacenar en el array es el valor 2^{31}-2.

Ejercicio 6.8:

Escriba un método, de nombre obtenerSumaArray, que reciba por parámetro un array de enteros y devuelva la suma de sus elementos.

Planteamiento: Se recorre el array para sumar sus valores en una variable que será devuelta al final del método.

Parámetros: El método recibe por parámetro un array de números enteros.

Valor de retorno: El método devuelve el resultado de sumar los elementos del array recibido. Como el array contiene números enteros, la suma de los mismos será un valor entero. Por tanto, el valor devuelto será del tipo `int`.

Solución:

```
public int sumaElementosArray(int[] arr){
    int suma = 0;

    for (int num : arr){
        suma += num;   <------   Se van acumulando los valores del array.
    }
    return suma;
}
```

Ejercicio 6.9:

Escriba un método, de nombre arrayPotencias2, que cree un array y lo rellene con potencias de 2. Las potencias de 2 comenzarán en 2^0 y el número total de ellas se recibirá por parámetro. El método devolverá el array creado.

Planteamiento: En primer lugar hay que crear un array para albergar tantos elementos como indique el número de potencias recibido por parámetro. A continuación, se guardarán en las posiciones del array las distintas potencias recorriendo el array con un bucle. Para conseguir que cada posición almacene el valor correspondiente a 2 elevado a su índice, se utilizará una variable que se va multiplicando por 2 en cada iteración del bucle. Finalmente se devuelve el array creado y relleno.

Parámetros: Número de potencias de 2 que almacenará el array. Si se recibe, por ejemplo, el valor 10, significa que el array almacenará los 10 valores correspondientes a las potencias de 2 comprendidas entre 2^0 y 2^9, ambas inclusive.

Valor de retorno: El método devolverá el array de enteros creado con las potencias de 2. Por tanto, el método devolverá un objeto del tipo `int[]`.

Especificación del valor de retorno del método como array de enteros.

Solución:

```
public int[] arrayPotencias2(int numPotencias){
    int[] potencias2 = new int[numPotencias];
    for (int i = 0, potencia = 1; i < potencias2.length; i++, potencia *= 2){
        potencias2[i] = potencia;
    }
    return potencias2;
}
```

Declaración y creación del array.

Aviso: Resultaría de interés que el método controle que el valor recibido por parámetro sea mayor o igual que 0 ya que, en otro caso no será posible crear el array y el método lanzará una excepción. Dado el rango de valores del tipo `int`, la mayor potencia de 2 que es posible almacenar en un array de enteros es 2^{30}, ya que 2^{31} supera en uno el rango del tipo.

Ejercicio 6.10:

Escriba un método, de nombre arrayMultiplicado, que reciba por parámetro un array de enteros y un factor de multiplicación. El método devolverá un nuevo array cuyos elementos serán los contenidos en el array recibido multiplicados por el factor especificado.

Planteamiento: Se crea el nuevo array con el mismo tamaño que el recibido. A continuación, se recorre con un bucle los dos arrays para asignar en cada posición del nuevo el resultado de multiplicar la misma posición del array recibido por el factor.

Parámetros: Un array de enteros y un factor para la multiplicación también entero.

Valor de retorno: El método devolverá el array creado y el resultado de multiplicar cada posición del antiguo por el factor. Como el array de entrada es un array de enteros, el array devuelto es también un array de enteros, es decir un objeto del tipo `int[]`.

Solución:

```
public int[] arrayMultiplicado(int[] arr, int factor){
    int[] nuevo = new int[arr.length];
    for (int i = 0; i < arr.length; i++){
```

Declaración y creación del nuevo array a partir de la longitud del recibido.

```
            nuevo[i] = arr[i] * factor;
        }
        return nuevo;
    }
```

Aviso: Hay que tener cuidado con los valores permitidos para el tipo `int`. Al multiplicar por un factor puede superar el límite admisible y dar resultados no esperados.

Ejercicio 6.11:

Escriba un método que reciba como parámetro un array de cadenas y devuelva la cadena resultante de concatenar todas las contenidas en el array.

Planteamiento: Tras comprobar que el array no es una referencia nula, el método declara y construye una nueva cadena vacía. A continuación se recorren las posiciones del array para ir concatenando las cadenas referenciadas por sus posiciones a la nueva cadena. Antes de proceder a la operación de concatenación, se comprueba que la correspondiente posición del array no es una referencia nula. Finalmente, se devuelve la cadena resultante.

Parámetros: El método recibe como parámetro un array de cadenas de caracteres u objetos de la clase `String`.

Valor de retorno: El método devuelve la cadena resultante de concatenar todas las cadenas referenciadas desde las posiciones del array recibido, es decir, un objeto de la clase `String`.

Solución:

```
public String arrayACadena(String[] arrCad){
    if(arrCad == null) <---------------------- Se comprueba que el parámetro no valga null.
        throw new IllegalArgumentException("Argumento no válido.");
    String nuevaCadena = "";
    for(String cad: arrCad){
        if(cad != null){
            nuevaCadena += cad; <------------- Se concatenan las cadenas del array a la nueva cadena.
        }
    }
    return nuevaCadena;
}
```

Aviso: Debería de tener cuidado con los valores que se reciben antes de utilizarlos en el método. En el Capítulo 4 se trataron con detalle las excepciones.

Ejercicio 6.12:

Escriba un método, de nombre obtenerSumaLongCadArray, que reciba por parámetro un array de cadenas y devuelva el número total de caracteres de todas las cadenas del array.

Planteamiento: Tras comprobar que el array recibido no es una referencia nula, se recorren todas sus posiciones. Se van sumando los tamaños de todas las cadenas y al final se devuelve la suma total. El tamaño de cada cadena se obtiene por el método `length()`. Antes de invocar este método se comprobará que no se trata de una referencia nula.

Parámetros: El método recibe como parámetro un array de cadenas de caracteres u objetos de la clase `String`.

Valor de retorno: El método devuelve la suma de todos los tamaños de todas las cadenas del array. Como la suma de tamaños es un valor entero el método devuelve un valor del tipo `int`.

Solución:

```
public int obtenerSumaLongCadArray(String[] arrCad){
    if(arrCad == null)
        throw new IllegalArgumentException("Argumento no válido");
    int longitudTotal = 0;
    for(String cad: arrCad){
        if(cad != null){
            longitudTotal += cad.length(); // Se acumulan los tamaños de las cadenas del array sólo si exsiten.
        }
    }
    return longitudTotal;
}
```

Ejercicio 6.13:

Escriba un método, de nombre obtenerLongCadenas, que reciba por parámetro un array de cadenas y devuelva un array de enteros con los tamaños de las cadenas contenidas en el array.

Planteamiento: Tras comprobar que el array recibido no es una referencia nula, se crea un array de enteros del mismo tamaño que el array de cadenas recibido por parámetro. Se recorren las posiciones del array de cadenas para obtener el tamaño de cada una de ellas y asignar este valor a la correspondiente posición del array de enteros. Si el array de cadenas tiene alguna referencia nula en alguna de las posiciones, el método asigna el valor 0 a la correspondiente posición del array de enteros. Finalmente, se devuelve el array con los tamaños de las cadenas.

Parámetros: El método recibe como parámetro un array de cadenas de caracteres u objetos de la clase `String`.

Valor de retorno: El método devuelve un array de enteros cuyas posiciones almacenan el número de caracteres de las distintas cadenas almacenadas en el array recibido por parámetro, es decir, un objeto del tipo `int[]`.

Solución:

```
public int[] obtenerLongCadenas(String[] arrCad){
    if(arrCad == null)
        throw new IllegalArgumentException("Argumento no válido");
    int[] arrLengths = new int[arrCad.length];
    int i = 0; // Asigna el tamaño de la cadena i-ésima. Se incrementa la variable i para la siguiente cadena.
    for(String cad: arrCad){
        if(cad != null){
            arrLengths[i++] = cad.length();
        }else{
            arrLengths[i++] = 0; // Si no existe el String se deja el valor 0 como tamaño. Es necesario incrementar i para mantener la correspondencia con las posiciones de las cadenas.
        }
    }
    return arrLengths;
}
```

Ejercicio 6.14:

Escriba una clase de nombre PruebaMetodosArrCad y añada, como método estático, el método realizado en el ejercicio anterior. Incluir en el método main() de la clase las sentencias necesarias para comprobar el correcto funcionamiento de dicho método.

Planteamiento: Tras añadir el método `obtenerLongCadenas` como método estático, se declara e inicializa en el método `main()` de la clase un array con 5 cadenas de caracteres. A continuación, se declara una referencia a un

array de enteros y se recoge el valor devuelto por el método obtenerLongCadenas() al que se especifica como argumento el anterior array de cadenas de caracteres.

Solución:

```
import java.util.ListIterator;
import java.util.Arrays;
import java.util.List;
public class PruebaMetodosArrCad {
      public static int[] obtenerLongCadenas(String[] arrCad){ } //como antes
      public static void main(String args[]){
          String[] arrCad1 = { null,
                          new String(""),
                          new String("a"),
                          new String("bb"),  <---- Se declara un array de cadenas y se asigna valor, simultáneamente, a sus posiciones.
                          new String("ccc"),
                          new String("dddd") };
          int[] longCadenas = obtenerLongCadenas(arrCad1);
          System.out.println("Longitudes de las cadenas: ");
          for(int lon : longCadenas){
              System.out.println(lon);
          }
      }
  }
```

Comentario: El resultado que debe aparecer por pantalla es:

```
0
0
1
2
3
4
```

Ejercicio 6.15:

Escriba un método, de nombre obtenerCadenaMasLarga, que reciba por parámetro un array de cadenas y devuelva la que contenga el mayor número de caracteres.

Planteamiento: En primer lugar se comprueba que el array recibido no es una cadena nula. A continuación se declaran dos variables, lonCadenaMasLarga y cadenaMasLarga que almacenan, respectivamente, la longitud de la mayor cadena encontrada y la cadena en concreto. Inicialmente se hace que la primera variable se inicialice a 0 y la segunda a null. Con un bucle se recorre secuencialmente las posiciones del array para comprobar si la cadena ubicada en la siguiente posición tiene una longitud mayor que todas las posiciones anteriores. En ese caso, las variables lonCadenaMasLarga y cadenaMasLarga se actualizan convenientemente. Finalmente, se devuelve la cadena referenciada por la variable cadenaMasLarga. Antes de obtener la longitud de cada cadena se comprueba que no se trate de una referencia nula.

Parámetros: El método recibe como parámetro un array de cadenas de caracteres u objetos de la clase String.

Valor de retorno: El método devuelve la cadena con mayor número de caracteres de entre todas las contenidas en el array recibido, es decir, un objeto de la clase String.

Solución:

```
public String obtenerCadenaMasLarga(String[] arrCad){
    if(arrCad == null)
        throw new IllegalArgumentException("Argumento no válido");
    String cadenaMasLarga = null;
    int longCadenaMasLarga = 0;
    for(String cad: arrCad){
        if(cad != null){
            if(cad.length() > longCadenaMasLarga){
                cadenaMasLarga = cad;
                longCadenaMasLarga = cad.length(); <-- Si la cadena en la posición i es de mayor longitud, ésta pasa a ser la cadena más larga.
            }
        }
    }
    return cadenaMasLarga;
}
```

Comentario: En lugar de 0 y `null`, las variables `lonCadenaMasLarga` y `cadenaMasLarga` podían haberse inicializado, respectivamente, con la longitud de la primera cadena del array y dicha cadena. El bucle, entonces, debería haber partido de la posición 1 del array. Fíjese que esto sólo es posible si el array no es de tamaño 0.

Ejercicio 6.16:

Escriba un método, de nombre obtenerArrCad5Vocales, que reciba por parámetro un array de cadenas y devuelva un array con las que contengan las 5 vocales. Para la consideración de un carácter como vocal no se tendrá en cuenta si está en mayúsculas o en minúsculas.

Planteamiento: En primer lugar se comprueba que el array recibido no es una cadena nula. Se procede, a continuación, a declarar y construir un array auxiliar con el mismo tamaño que el array recibido, ya que es posible que todas sus cadenas cumplan la condición de contener las 5 vocales. Se recorre el array recibido para comprobar si cada una de sus cadenas tiene los caracteres correspondientes a las 5 vocales. Esta comprobación se realiza con la ayuda del método `indexOf()` de la clase `String`. Cuando la comprobación es positiva, la cadena se guarda en el array auxiliar y, al mismo tiempo, se actualiza un contador con el número de cadenas encontradas que satisfacen la condición. El siguiente paso es construir un nuevo array para almacenar exactamente el número de cadenas encontradas. Esta operación se realiza con ayuda del valor del contador utilizado en el bucle previo. El último bucle se encarga de copiar las cadenas de 5 vocales desde el array auxiliar al último array creado que, finalmente, se devuelve.

Parámetros: El método recibe como parámetro un array de cadenas de caracteres u objetos de la clase `String`.

Valor de retorno: El método devuelve un array con las cadenas presentes en el array recibido que contienen las 5 vocales, es decir, un objeto de la clase `String[]`.

Solución:

```
public String[] obtenerArrCad5Vocales(String[] arrCad){
    if(arrCad == null)
        throw new IllegalArgumentException("Argumento no válido.");
    String[] arrayAux = new String[arrCad.length];
    int numCadenas5Vocales = 0;
    int j = 0;
    for(String cad: arrCad){
        if (cad != null){
```

```
            String cadAux = cad.toUpperCase();
            if (cadAux.indexOf('A') != -1 &&
                cadAux.indexOf('E') != -1 &&
                cadAux.indexOf('I') != -1 &&
                cadAux.indexOf('O') != -1 &&
                cadAux.indexOf('U') != -1){
                    arrayAux[j++] = cad;
                    numCadenas5Vocales++;
            }
        }
    }

    if(numCadenas5Vocales != 0){
        String[] arrCad5Vocales = new String[numCadenas5Vocales];
        for(int i = 0; i < numCadenas5Vocales; i++){
            arrCad5Vocales[i] = arrayAux[i];
        }
        return arrCad5Vocales;
    }else{
        return null;
    }
}
```

Se utiliza esta cadena auxiliar para no llamar al método *toUpperCase()* para la comprobación de cada vocal.

indexOf() devuelve -1 si no se encuentra el carácter.

Se crea un nuevo array para que el array devuelto tenga el mismo tamaño que el número de cadenas con las 5 vocales.

Comentario: Una estrategia a considerar es que, en lugar de devolver `null` cuando el array del parámetro no tiene ninguna cadena con las cinco vocales, devolver un array de 0 elementos. El código queda más limpio y consistente, sin casos especiales.

Ejercicio 6.17:

Escriba un método, de nombre obtenerNumVecesCar, que reciba por parámetro un array de cadenas y un carácter. El método devolverá el número de veces que el carácter especificado figura en todas las cadenas contenidas en el array.

Planteamiento: Una vez comprobado que el array recibido no es una referencia nula, se recorren todas sus cadenas para ir obteniendo el número de veces que el carácter especificado figura en ellas. Además del bucle que recorre las posiciones del array recibido, se incluye otro bucle que pasa por todos los caracteres de cada una de las cadenas. Cada vez que se detecte la ocurrencia del carácter especificado en una de las cadenas se incrementará una variable contador.

Parámetros: El método recibe como parámetro un array de cadenas de caracteres u objetos de la clase `String` y el carácter (`char`) a buscar.

Valor de retorno: El método devuelve, como valor entero, el número total de ocurrencias del carácter especificado en todas las cadenas del array recibido por parámetro, es decir, un valor del tipo `int`.

Solución:

```
public int obtenerNumVecesCar(String[] arrCad, char car){
    if(arrCad == null)
        throw new IllegalArgumentException("Parametro no válido.");
    int numVeces = 0;
    for(String cad: arrCad){
        if(cad != null){
```

```
            for(int i = 0; i < cad.length(); i++){
                if(cad.charAt(i) == car){
                    numVeces++;
                }
            }
        }
    }
    return (numVeces);
}
```

Se cuenta uno más cada vez que se encuentra.

Ejercicio 6.18:

Escriba un método, de nombre obtenerArrayOrdAlfab, que reciba por parámetro un array de cadenas de caracteres y las ordene alfabéticamente. La ordenación no se verá afectada por la expresión de los caracteres expresados en mayúsculas o minúsculas. Es decir, las cadenas "ALBACETE", "antonio", y "BURGOS" quedarán ordenadas en este mismo orden.

Planteamiento: El método comprueba que el array recibido no es una referencia nula. A continuación, se recorre cada una de las posiciones del array recibido. Para cada una de ellas se hace un nuevo recorrido comenzando desde la posición siguiente hasta el final del array. Si se encuentra que alguna de las posiciones siguientes almacena una cadena alfabéticamente menor se procede al intercambio de posiciones.

Parámetros: El método recibe como parámetro un array con las cadenas de caracteres que se ordenarán alfabéticamente.

Valor de retorno: El método no necesita devolver ningún valor de retorno, por lo que se utiliza la cláusula `void`.

Solución:

```
public void obtenerArrayOrdAlfab(String[] arrCad){
    if(arrCad == null)
        throw new IllegalArgumentException("Parametro no válido.");
    int numVeces = 0;
    for(int i = 0; i < arrCad.length; i++){
        if(arrCad[i] != null){
            for(int j = i+1; j < arrCad.length; j++){
                if(arrCad[j] != null &&
                   arrCad[j].toUpperCase().compareTo(
                   arrCad[i].toUpperCase()) < 0 ){
                    String cadAux = arrCad[i];
                    arrCad[i] = arrCad[j];
                    arrCad[j] = cadAux;
                }
            }
        }
    }
}
```

Para cada cadena del array se comprueba si todas las siguientes son menores alfabéticamente.

En caso de encontrar una cadena alfabéticamente menor, se intercambian para que el array vaya quedando ordenado.

Comentario: La ordenación de datos de arrays se puede utilizar utilizando la API de la clase `Arrays`, como se verá en ejercicios más adelante en este mismo capítulo. De hecho, la llamada a este método se puede sustituir por:

```
Arrays.sort(arrCad, String.CASE_INSENSITIVE_ORDER);
```

ARRAYS MULTIDIMENSIONALES

Ejercicio 6.19:

Escriba un método, de nombre rellenarMatrizSecuencia1D, que reciba una matriz de enteros por parámetro y la rellene para que sus posiciones almacenen un valor que se irá incrementando en una unidad por filas. La matriz se rellenará de manera que dos elementos consecutivos según la segunda dimensión almacenen dos valores también consecutivos. Una matriz de 5 elementos en la primera dimensión y 5 en la segunda quedaría como sigue:

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24

Planteamiento: Una vez comprobado que el array bidimensional recibido no constituye una referencia nula, se recorren todas sus posiciones haciendo que se acceda a todos los elementos primero por su índice según la primera dimensión y luego por el de la segunda. Cada vez que se da valor a una posición, éste se incrementará para su asignación a la posición siguiente.

Parámetros: El método recibe como parámetro un array bidimensional (matriz) de enteros para asignar valores a sus posiciones.

Valor de retorno: El método no devuelve ningún valor de retorno ya que se modifican directamente las posiciones del propio array bidimensional recibido como entrada. Como el método no necesita devolver ningún valor de retorno se utiliza la cláusula `void`.

Solución:

```
public void rellenarMatrizSecuencial1D(int[][] matriz){
    if(matriz == null){
        throw new IllegalArgumentException("Parametro no válido.");
    }
    for(int i = 0, k = 0; i < matriz.length; i++){
        for(int j = 0; j < matriz[i].length; j++)
            matriz[i][j] = k++;
    }
}
```

Se asigna el valor correspondiente a cada posición y luego se incrementa.

Ejercicio 6.20:

Escriba un método, de nombre rellenarMatrizSecuencia2D, que reciba una matriz de enteros por parámetro y la rellene para que sus posiciones almacenen un valor que se irá incrementando en una unidad por columnas. La matriz se rellenará de manera que dos elementos consecutivos según la primera dimensión almacenen dos valores también consecutivos. Una matriz de 5 elementos en la primera dimensión y 5 en la segunda quedaría como sigue:

0	5	10	15	20
1	6	11	16	21
2	7	12	17	22
3	8	13	18	23
4	9	14	19	24

Planteamiento: Una vez comprobado que el array bidimensional recibido no se trata de una referencia nula, se recorren todas sus posiciones haciendo que se acceda a todos los elementos primero por su índice según la segunda dimensión y luego por el de la primera. Cada vez que se da valor a una posición, éste se incrementará para su asignación a la posición siguiente.

Parámetros: El método recibe como parámetro un array bidimensional (matriz) de enteros para asignar valores a sus posiciones.

Valor de retorno: El método no devuelve ningún valor de retorno ya que se modifican directamente las posiciones del propio array bidimensional recibido como entrada. Como el método no necesita devolver ningún valor de retorno se utiliza la cláusula void.

Solución:

```
public void rellenarMatrizSecuencial2D(int[][] matriz){
    if(matriz == null){
        throw new IllegalArgumentException("Parametro no válido.");
    }
    for(int j = 0, k = 0; j < matriz[0].length; j++){
        for(int i = 0; i < matriz.length; i++)
            matriz[i][j] = k++;
    }
}
```

Se supone que todas las columnas tienen el mismo número de elementos que la primera de ellas.

Ejercicio 6.21:

Escriba un método, de nombre obtenerMatrizIDentidad, que reciba por parámetro un valor entero para especificar el número de elementos para las dos dimensiones de la matriz. El método creará la correspondiente matriz y rellenará sus posiciones para que constituya la matriz identidad de la dimensión especificada. Indicar que una matriz identidad es una matriz cuadrada donde los elementos de la diagonal principal tienen valor 1 y el resto de elementos tiene valor 0. Si el método recibe por parámetro el valor 5, la matriz identidad devuelta sería como la siguiente:

1	0	0	0	0
0	1	0	0	0
0	0	1	0	0
0	0	0	1	0
0	0	0	0	1

Planteamiento: El método crea una matriz cuadrada con el número indicado de elementos para cada dimensión como especifique el parámetro recibido. Se utiliza un único bucle para acceder y dar valor a las posiciones correspondientes a la diagonal principal de la matriz, ya que todas ellas verifican que su índice por la primera dimensión es igual al de la segunda dimensión. El resto de posiciones de la matriz no necesitan ser modificadas ya que al crear una matriz de enteros todos sus elementos se inicializan a su valor por defecto (0).

Parámetros: Valor entero que especifica el número de elementos en las dos dimensiones de la matriz. Dado que todas las matrices identidad han de ser cuadradas basta con recibir un solo valor.

Valor de retorno: El método devuelve la matriz identidad construida como array bidimensional de enteros, es decir, devuelve un objeto del tipo int[][].

Solución:

```
public int[][] obtenerMatrizIdentidad(int dimension){
```

```
        if(dimension <= 0 ){
            throw new IllegalArgumentException("Parametro no válido.");
        }

        int[][] matriz = new int[dimension][dimension];
        for(int i = 0; i < dimension; i++){
            matriz[i][i] = 1;
        }
        return matriz;
    }
```

Sólo es necesario dar valor a los elementos de la diagonal principal.

Ejercicio 6.22:

Escriba un método, de nombre mostrarMatriz1D, que reciba por parámetro un array bidimensional (matriz) de enteros y muestre sus elementos por pantalla de forma que la primera dimensión de la matriz se corresponda con las filas y la segunda con las columnas.

Planteamiento: En primer lugar se comprueba que la referencia recibida no es nula. A continuación se recorre la matriz accediendo a sus elementos de forma que para cada valor del índice según la primera dimensión se recorran todos los valores del índice según la segunda.

Para conseguir un mejor efecto visual, se hace que todos los elementos de la matriz se representen por dos caracteres y vayan seguidos de un espacio en blanco. En el caso de valores de una sola cifra, se añade un espacio adicional a su derecha para mantenerlos alineados con los de dos cifras. Al terminar de mostrar los elementos correspondientes a cada fila, será necesario insertar un salto de línea para mostrar la siguiente fila en la línea siguiente.

Parámetros: Array bidimensional (matriz) de enteros cuyos elementos se mostrarán por pantalla.

Valor de retorno: El método no necesita devolver ningún valor de retorno por lo que se utiliza la cláusula `void`.

Solución:

```
public void mostrarMatrizDim1(int[][] matriz){
    if(matriz == null){
        throw new IllegalArgumentException("Parametro no válido.");
    }
    for(int i = 0; i < matriz.length; i++){
        for(int j = 0; j<matriz[i].length; j++){
            System.out.print(matriz[i][j] +
                         (matriz[i][j]<10 == 0 ? " " : "")
                         + " ");
        }
        System.out.println();
    }
}
```

Se añade un espacio blanco a los elementos de una sola cifra.

Salto de línea al terminar de mostrar los elementos de una fila.

Comentario: Para escribir con formato y que queden alineados es preferible utilizar el método `printf` de la siguiente forma:

```
System.out.printf("%2d ", matriz[i][j]);
```

o con un valor mayor de espacios.

Ejercicio 6.23:

Escriba un método, de nombre matrizCharComoString, que reciba por parámetro un array bidimensional (matriz) de caracteres y devuelva una cadena con la representación textual de la matriz recibida. La cadena dispondrá los elementos de la matriz de forma que la primera dimensión se corresponda con las filas y la segunda con las columnas.

Planteamiento: En primer lugar se comprueba que la referencia recibida no es nula. A continuación se recorre la matriz accediendo a sus elementos de forma que para cada valor del índice según la primera dimensión se recorran todos los valores del índice según la segunda. Los elementos se concatenan a una cadena que finalmente se devuelve. Al terminar de incluir los elementos correspondientes a cada fila, se insertará en la cadena un salto de línea para separarlos de los elementos de la fila siguiente.

Parámetros: Array bidimensional (matriz) de caracteres cuyos elementos se devolverán como una cadena.

Valor de retorno: Se devuelve una cadena con la representación textual de la matriz recibida por parámetro, es decir, un objeto de la clase `String`.

Solución:

```
public String matrizCharComoString(char[][] matriz){
   if(matriz == null){
       throw new IllegalArgumentException("Parametro no válido.");
   }
   String resul = "";
   for(int i = 0; i < matriz.length; i++){
       for(int j = 0; j < matriz[i].length; j++){
           resul += (matriz[i][j] + " "); <-- Los caracteres de la matriz se concatenan a la cadena que hay que devolver.
       }
       resul += "\n";
   }
   return resul;
}
```

Ejercicio 6.24:

Escriba un método, de nombre matrizIntComoString, que reciba por parámetro un array bidimensional (matriz) de enteros y devuelva una cadena con la representación textual de la matriz recibida. La cadena dispondrá los elementos de la matriz de forma que la primera dimensión se corresponda con las filas y la segunda con las columnas.

Planteamiento: En primer lugar se comprueba que la referencia recibida no es nula. A continuación se recorre la matriz accediendo a sus elementos de forma que para cada valor del índice según la primera dimensión se recorran todos los valores del índice según la segunda. Los elementos son concatenados a una cadena que finalmente será devuelta. Al terminar de mostrar los elementos correspondientes a cada fila, se insertará en la cadena un salto de línea para separarlos de los elementos de la fila siguiente.

Parámetros: Array bidimensional (matriz) de enteros cuyos elementos se devolverán como una cadena.

Valor de retorno: Se devuelve una cadena con la representación textual de la matriz recibida por parámetro, es decir, un objeto de la clase `String`.

Solución:

```
public String matrizIntComoString(int[][] matriz){
```

```
        if(matriz == null){
            throw new IllegalArgumentException("Parametros no válido.");
        }
        String resul = "";
        for(int i = 0; i < matriz.length; i++){
            for(int j = 0; j < matriz[i].length; j++){
                resul += (matriz[i][j] + (matriz[i][j]<10 == 0 ? " " : "") + " ");
            }
            resol += "\n";
        }
        return resul;
    }
```

Los valores enteros de la matriz se concatenan a la cadena a devolver. Los enteros de una sola cifra se concatenan precedidos de un espacio en blanco.

Comentario: Para conseguir ir creando el formato establecido se puede utilizar un objeto de la clase Formatter, cuya sintaxis es como la del método System.out.printf(). Véase como utilizarlo en el Ejercicio 5.1.

Ejercicio 6.25:

Escriba un método, de nombre rellenarMatrizAsteriscos, que reciba por parámetro un valor entero que especificará el número de filas de asteriscos que albergará la matriz. La primera fila contendrá un solo asterisco situado en la posición central según la segunda dimensión de la matriz. Cada nueva fila contendrá dos asteriscos más y también se encontrarán centrados según la segunda dimensión de la matriz. El aspecto final que debe presentar la matriz si se recibe un número de filas de asteriscos sería el siguiente:

				*				
			*	*	*			
		*	*	*	*	*		
	*	*	*	*	*	*	*	
*	*	*	*	*	*	*	*	*

Planteamiento: En primer lugar se comprueba que el número de filas recibido es mayor que 0. A continuación se construye una matriz con las dimensiones apropiadas. Para ello se tiene en cuenta que la disposición de los asteriscos en forma piramidal exige que el número de columnas sea el resultado de la operación 2*numFilas-1. Así se consigue una matriz con la misma cantidad de elementos en la primera dimensión que el número de filas de asteriscos especificados y con el número exacto de elementos en la segunda dimensión (columnas) para que las filas de asteriscos queden centradas. Se utiliza un bucle para ir asignando los asteriscos en las posiciones correspondientes de la matriz. El cuerpo de este bucle está divido en tres partes, correspondientes a tres bucles que, respectivamente, asignan a cada fila de la matriz los caracteres blancos que preceden a los asteriscos, los propios asteriscos y los caracteres blancos que siguen a los asteriscos. Finalmente se devuelve la matriz creada.

Parámetros: Valor entero que especifica el número de filas de asteriscos que albergará la matriz.

Valor de retorno: Se devuelve una matriz de elementos de tipo char rellena con asteriscos de la forma especificada, es decir, un objeto del tipo char[][].

Solución:

```
public char[][] rellenarMatrizAsteriscos(int numFilas){
    if(numFilas <= 0){
        throw new IllegalArgumentException("Parametro incorrecto");
    }
    char[][] matrizAst = new char[numFilas][numFilas*2-1];
```

```
    for(int i = 0, numAsteriscos = 1, posInicio = matrizAst[0].length/2;
        i < numFilas;
        i++,numAsteriscos += 2, posInicio--){
        for(int j = 0; j<posInicio;j++){
            matrizAst[i][j] = " ";  <-- Posiciones con caracteres blancos antes de los asteriscos.
        }
        for(int j = 0, pos = posInicio; j<numAsteriscos; j++){
            matrizAst[i][pos++] = "*";  <-- Posiciones con asteriscos.
        }
        for(int j = posInicio+numAsteriscos; j<matrizAst[i].length; j++){
            matrizAst[i][j] = " ";  <-- Posiciones con caracteres blancos después de los asteriscos.
        }
    }
    return matrizAst;
}
```

Comentario: El bucle principal mantiene 3 variables para ir actualizando el número de asteriscos de cada fila (`numAsteriscos`), la posición respecto a la segunda dimensión a partir de la que deben escribirse los asteriscos en cada fila (`posInicio`) y el propio número de filas de asteriscos que albergará la matriz (`numFilas`). El número de asteriscos se incrementa en dos unidades porque cada fila añadirá dos asteriscos más. La posición de inicio para la escritura de asteriscos en cada fila se decrementa en una unidad porque sólo uno de los dos nuevos asteriscos precede al resto, el otro se colocará al final.

Ejercicio 6.26:

Escriba un método, de nombre obtenerSumaElementosMatriz, que reciba por parámetro un array bidimensional de números enteros y devuelva la suma de todos sus elementos.

Planteamiento: Después de comprobar que la referencia recibida no es nula, se recorren todos los elementos de la matriz para sumar su valor a una variable. Finalmente la variable es devuelta.

Parámetros: Array bidimensional (matriz) para obtener la suma de sus elementos.

Valor de retorno: Se devuelve un valor entero correspondiente a la suma de los elementos de la matriz recibida por parámetro. Como la matriz es una matriz de números enteros se devuelve un valor del tipo `int`.

Solución:

```
public int obtenerSumaElementosMatriz(int matriz[][]){

    if(matriz == null){
        throw new IllegalArgumentException("Argumento no valido");
    }
    int suma = 0;
    for(int i = 0; i < matriz.length; i++){
        for(int j = 0; j<matriz[i].length; j++){
            suma += matriz[i][j];  <-- Los elementos de la matriz se van acumulando en la variable suma.
        }
    }
    return suma;

}
```

Ejercicio 6.27:

Escriba un método, de nombre obtenerMatrizProducto, que reciba por parámetro dos matrices de valores enteros y devuelva la matriz resultante de multiplicar matricialmente las dos recibidas por parámetro.

Planteamiento: En primer lugar se comprueba que las referencias a las dos matrices recibidas no son nulas y que sus dimensiones permiten su multiplicación, para ello debe verificarse que el número de elementos de la segunda dimensión (columnas) de la primera matriz es igual al número de elementos de la primera dimensión de la segunda.

A continuación, se construye la matriz resultado teniendo en cuenta que sus dimensiones corresponderán al número de filas de la primera matriz y al número de columnas de la segunda. Se utilizan tres bucles para calcular los elementos de la matriz producto. Los dos bucles más externos recorren las posiciones de la matriz resultado. El bucle más interno es necesario ya que cada elemento de la matriz producto debe obtenerse mediante la suma de los productos de los elementos de una fila de la primera matriz por los elementos de una columna de la segunda. Finalmente, se devuelve la matriz producto.

Parámetros: Dos arrays bidimensionales correspondientes a las matrices que se van a multiplicar

Valor de retorno: Se devuelve una matriz de enteros correspondiente a la matriz resultado de la multiplicación de las dos matrices recibidas por parámetro. Como las dos matrices de los parámetros son matrices de números enteros, la matriz resultado de la multiplicación es un objeto del tipo `int[][]`.

Solución:

```
public int[][] obtenerMatrizProducto(int m1[][], int[][] m2){
    if((m1 == null || m2 == null) || (m1[0].length != m2.length)){
        throw new IllegalArgumentException("Parametros incorrectos.");
    }
    int[][] matrizProducto = new int[m1.length][m2[0].length];
    for(int i = 0; i < matrizProducto.length; i++){
        for(int j = 0; j < matrizProducto[i].length; j++){
            for(int k = 0; k < m2.length; k++){
                matrizProducto[i][j] += m1[i][k] * m2[k][j];
            }
        }
    }
    return matrizProducto;
}
```

Ejercicio 6.28:

Escriba un método, de nombre obtenerDiagonales, que reciba por parámetro una matriz de valores enteros y devuelva una matriz con los valores de las dos diagonales principales de la matriz recibida.

Planteamiento: En primer lugar se comprueba que la referencia a la matriz recibida no sea nula y que corresponde a una matriz cuadrada. A continuación se construye la matriz para albergar los valores correspondientes a las dos diagonales.

Esta matriz tendrá 2 elementos en su primera dimensión y el número de filas o de columnas (serán iguales) de la matriz recibida por parámetro en la segunda. Basta un único bucle para avanzar a la vez por los elementos de las dos diagonales que se irán almacenando en su correspondiente posición de la matriz a devolver. Finalmente se devuelve la matriz con los elementos de las dos diagonales.

Parámetros: Un array bidimensional correspondiente a las matriz de la que se van a obtener los elementos de sus diagonales.

Valor de retorno: Se devuelve una matriz de enteros con los elementos correspondientes a las dos diagonales de la matriz recibida. Como la matriz del parámetro es una matriz de números enteros se devuelve un objeto del tipo `int[][]`.

Solución:

```
public int[][] obtenerDiagonales(int[][] matriz){
    if(matriz == null || matriz.length != matriz[0].length){
        throw new IllegalArgumentException("Argumento no valido");
    }
    int[][] diagonales = new int[2][matriz.length];
    for(int i = 0,j = 0; i < matriz.length; i++, j++){
        diagonales[0][j] = matriz[i][j];
        diagonales[1][j] = matriz[matriz.length-1-i][j];
    }
    return diagonales;
}
```

La variable i se utiliza para recorrer las filas y la variable j para las columnas.

Ejercicio 6.29:

Escriba un programa que obtenga y muestre por pantalla una matriz de enteros con el siguiente aspecto.

1	2	3	4	5
1	2	3	4	
1	2	3		
1	2			
1				

Planteamiento: Se crea la matriz especificando para cada elemento de la primera dimensión un array de enteros con distinto número de elementos. A continuación, se muestra por pantalla la matriz creada.

Solución:

```
public class MatrizNoRegular1{
    public static void main(String args[]){
        int[][] matrizInt = {{1,2,3,4,5}, {1,2,3,4}, {1,2,3}, {1,2}, {1}};
        for(int i = 0; i < matrizInt.length; i++){
            for(int j = 0; j<matrizInt[i].length; j++){
                System.out.print(matrizInt[i][j]+ " ");
            }
            System.out.println();
        }
    }
}
```

Ejercicio 6.30:

Escriba un programa que obtenga y muestre por pantalla una matriz de cadenas de caracteres con el siguiente aspecto. Indicar que el único elemento de la primera fila contiene una cadena con una 'A', los dos elementos de la segunda fila contienen, cada uno, una cadena con dos 'A', los de la tercera fila contiene cadenas con tres 'A' y así sucesivamente.

A		
AA	AA	
AAA	AAA	AAA

AAAA	*AAAA*	*AAAA*	*AAAA*	
AAAAA	*AAAAA*	*AAAAA*	*AAAAA*	*AAAAA*

Planteamiento: Se crea una matriz de objetos que sólo especifica el número de elementos (5, para la resolución del ejercicio) para la primera dimensión. Mediante un bucle se recorre la matriz para ir asignando a cada posición de la primera dimensión un array de objetos `String` con un elemento más cada vez . Los dos últimos bucles recorren la matriz para ir asignando las cadenas a sus posiciones. Se utiliza una cadena que es asignada a todos los elementos de la segunda dimensión. Al concluir se concatena una 'A' a la cadena para que se asigne a los elementos de la siguiente fila. Finalmente, se muestra la matriz por pantalla.

Solución:

```
public class MatrizNoRegular2{
    public static void main(String args[]){
        String[][] matrizStr;
        matrizStr = new String[5][];
        for(int i = 0; i < matrizStr.length; i++){
            matrizStr[i] = new String[i+1];
        }
        String cadA = "A";
        for(int i = 0; i < matrizStr.length; i++){
            for(int j = 0; j < matrizStr[i].length; j++){
                matrizStr[i][j] = cadA;
            }
            cadA += "A";
        }
        for(int i = 0; i < matrizStr.length; i++){
            for(int j = 0; j < matrizStr[i].length; j++){
                System.out.print(matrizStr[i][j]+ " ");
            }
            System.out.println();
        }
    }
}
```

Ejercicio 6.31:

Escriba un método, de nombre obtenerLaterales, que reciba por parámetro una matriz de valores enteros y devuelva una matriz con los valores de los cuatro laterales —superior, izquierdo, derecho e inferior—, de la matriz recibida.

Planteamiento: El método comienza comprobando que la referencia a la matriz recibida no sea nula. A continuación se construye la matriz que albergará los elementos de los cuatro laterales. Las dimensiones de esta matriz serán de 4 elementos en su primera dimensión, siendo el número de elementos de la segunda variable en función del número de filas y columnas de la matriz recibida por parámetro. Un primer bucle obtiene los elementos correspondientes a los laterales superior e inferior ya que ambos se disponen a lo largo de la segunda dimensión de la matriz. El segundo bucle hace lo mismo con los elementos de los laterales izquierdo y derecho porque para ambos es necesario recorrer la matriz por su primera dimensión. Por último se devuelve la matriz creada.

Parámetros: Un array bidimensional correspondiente a las matriz de la que se van a obtener los elementos de sus laterales.

Valor de retorno: Se devuelve una matriz de enteros que almacena los elementos de los cuatro laterales de la matriz recibida. Como la matriz del parámetro es una matriz de enteros se devuelve un objeto del tipo `int[][]`.

Solución:

```
public int[][] obtenerLaterales(int[][] matriz){
    if(matriz == null){
        throw new IllegalArgumentException("Parametros incorrectos");
    }
    int[][] laterales = {new int[matriz[0].length], //lateral Superior
                         new int[matriz.length],    //lateral Derecho
                         new int[matriz[0].length], //lateral Inferior
                         new int[matriz.length]};   //lateral Izquierdo

    for(int i = 0; i < matriz[0].length; i++){
        laterales[0][i] = matriz[0][i];
        laterales[2][i] = matriz[matriz.length-1][i];
    }
    for(int i = 0; i < matriz.length; i++){
        laterales[1][i] = matriz[i][matriz[0].length-1];
        laterales[3][i] = matriz[i][0];
    }
    return laterales;
}
```

Para los valores de los laterales superior e inferior.

Para los valores de los laterales izquierdo y derecho.

API DE MANEJO DE ARRAYS

Ejercicio 6.32:

Escriba una clase, de nombre APIManejoArrays1, cuyo método main() implemente un programa que lleve a cabo las siguientes acciones:

a) Declarar y construir un array de 30 enteros, de nombre arrInt1. Hacer que el array almacene en las 10 primeras posiciones 10 ceros, en las 10 siguientes 10 unos y en las 10 últimas de nuevo 10 ceros. Mostrar el contenido del array.

b) Declarar y construir un array de 6 caracteres, de nombre arrChar. El array almacenará en la primera mitad de las posiciones el carácter a y en la segunda mitad el carácter b. Mostrar el contenido del array.

c) Declarar y construir un array de 10 enteros, de nombre arrInt2. Hacer que el array almacene en sus posiciones los dígitos del 9 al 0 para mostrar después su contenido. Ordenar el array ascendentemente y mostrar de nuevo su contenido.

Planteamiento: Para la resolución del ejercicio se utilizarán algunos de los métodos que ofrece la case de utilidades `Arrays`. Con ayuda del método `fill()` es sencillo rellenar los arrays del modo especificado. En una de sus implementaciones este método recibe la posición de inicio y fin del intervalo a rellenar. Es importante tener en cuenta que la posición que especifica el fin no se incluye en la operación de relleno.

El método `sort()` ordena el array ascendentemente y se utiliza el método `toString()` para mostrar por pantalla el contenido de los arrays sin recurrir al uso de un bucle.

Solución:

```
public class APIManejoArrays1{
    public static void main(String[] args){
```

```
        int[] arrInt1 = new int[30];
        Arrays.fill(arrInt1,  0, 10, 0);
        Arrays.fill(arrInt1, 10, 20, 1);
        Arrays.fill(arrInt1, 20, 30, 0);
        System.out.println(Arrays.toString(arrInt1));

        char[] arrChar = new char[6];
        Arrays.fill(arrChar, 0, arrChar.length/2, 'a');
        Arrays.fill(arrChar, arrChar.length/2, arrChar.length, 'b');
        System.out.println(Arrays.toString(arrChar));

        int[] arrInt2 = new int[10];
        for(int i = 0; i < arrInt2.length; i++){
            arrInt2[i] = arrInt2.length - 1 - i;
        }
        System.out.println(Arrays.toString(arrInt2));
        Arrays.sort(arrInt2); <-- Se ordena el array con ayuda del método sort().
        System.out.println(Arrays.toString(arrInt2));

    }
}
```

Ejercicio 6.33:

Escriba una clase, de nombre APIManejoArrays1, cuyo método main() implemente un programa que lleve a cabo las siguientes acciones:

a) Declarar y construir un array de cadenas de objetos String de nombre arrStr que contenga las siguientes cadenas: "impresora","peto","mar","orilla" y "Orihuela"
b) Ordenar el array arrStr alfabéticamente teniendo en cuenta que las letras mayúsculas figuran en el código Unicode antes que las minúsculas. Mostrar por pantalla el resultado de la ordenación.
c) Ordenar el array arrStr alfabéticamente sin distinguir entre letras minúsculas y mayúsculas. Mostrar por pantalla el resultado de la ordenación.
d) Ordenar el array arrStr con un comparador que determine que una cadena con más caracteres precede a otra que tenga menos. Mostrar por pantalla el resultado de la ordenación.
e) Ordenar el array arrStr con un comparador que determine que una cadena con más vocales sigue a otra que tenga menos. Mostrar por pantalla el resultado de la ordenación.

Planteamiento: El array de cadenas es construido de forma estática inicializando directamente sus elementos. Se utilizará nuevamente el método `sort()` de la clase `Arrays` que puede utilizarse también para la ordenación de arrays de objetos. Para la resolución de los apartados b) y c) se utiliza el método sin especificar un elemento que defina un criterio de ordenación. De esta forma, se utiliza el *orden natural* establecido para los elementos del array que, por tratarse de cadenas de caracteres, es el orden alfabético. El orden natural considera las letras mayúsculas como anteriores a todas las minúsculas, situación que se puede modificar indicando que la ordenación no tenga en cuenta las diferencias entre letras mayúsculas y minúsculas.

El método `sort()` permite, además, realizar la ordenación basándose en un comparador que deberá especificarse y que permite determinar cuándo un objeto debe seguir a otro. Así, para resolver los apartados d) y e) se crean dos comparadores nuevos para los objetos de tipo `String`. El primero (`ComparadorCadLen`) considera que una cadena con más caracteres debe preceder a otra que tenga menos. Es decir, las cadenas quedarán ordenadas descendentemente por su longitud. El segundo (`ComparadorCadNumVocales`) determina que una cadena con más vocales debe seguir a otra cadena que tenga menos. O de otra forma, las cadenas quedarán ordenadas ascendentemente por el número de vocales que contienen. Para contabilizarlas no se descartarán las

vocales repetidas. Estos comparadores son previamente creados como clases que implementan la interfaz Comparator<String>.

El método sort() permite ordenar tanto arrays numéricos como de objetos. Para los segundos es posible hacerlo especificando un comparador que determine cuándo un objeto debe seguir a otro. En este caso se crean dos comparadores nuevos para los objetos de tipo String. El primero considera que una cadena con más caracteres debe preceder a otra que tenga menos. Es decir, las cadenas quedarán ordenadas descendentemente por su tamaño. El segundo determina que una cadena con más vocales debe seguir a otra cadena que tenga menos. Es decir, las cadenas quedarán ordenadas ascendentemente por el número de vocales que contienen. Para contabilizarlas no se descartarán las vocales repetidas. Estos comparadores se crean como clases que implementan la interfaz Comparator<String>.

Solución:

```
class ComparadorCadLen implements Comparator<String>{
      public ComparadorCadLen(){}
      public int compare(String a, String b){
          return b.length()-a.length();
      }
 }

class ComparadorCadNumVocales implements Comparator<String>{
   private int cuentaVocales(String cad){
          int numVocales = 0;
          for(int i = 0; i < cad.length(); i++){
              switch (cad.toUpperCase().charAt(i)){
                case 'A':
                case 'E':
                case 'I':
                case 'O':
                case 'U': numVocales++;
                          break;
              }
          }
      return numVocales;
   }

   public int compare(String a, String b){
      return cuentaVocales(a) - cuentaVocales(b);
   }

   public static void main (String[] args){
      String[] arrStr = {"impresora",
                         "peto",
                         "mar",
                         "orilla",
                         "Orihuela"};
      Arrays.sort(arrStr);
      System.out.println(Arrays.toString(arrStr));
      Arrays.sort(arrStr, String.CASE_INSENSITIVE_ORDER);
      System.out.println(Arrays.toString(arrStr));

      Arrays.sort(arrStr, new ComparadorCadLen());
      System.out.println(Arrays.toString(arrStr));
```

Es privado, pues sólo lo debe utilizar el método comparador.

Así la ordenación tiene en cuenta que las letras mayúsculas van en Unicode antes que las minúsculas.

Se ordena sin tener en cuenta mayúsculas y minúsculas.

Se ordenan con el comparador por la longitud de las cadenas.

```
            Arrays.sort(arrStr, new ComparadorCadNumVocales());
            System.out.println(Arrays.toString(arrStr));
        }
    }
```

Ejercicio 6.34:

Sea la clase Alumno que se presenta a continuación:

```
    public class Alumno {
        int númeroPersonal;
        String apellido1, apellido2, nombre;
        int numAsignaturas;
        double[] notasFinales;
        double notaMediaFinal;

        public Alumno(int numPer, String ap1, String ap2, String nom, int numAsig) {
            númeroPersonal = numPer;
            apellido1 = ap1;
            apellido2 = ap2;
            nombre = nom;
            numAsignaturas = numAsig;
            notasFinales = new double[numAsignaturas];
        }

        public Alumno(int numPer, String ap1, String ap2, String nom, int numAsig,
                      double[] notasF, double nmf) {
            númeroPersonal = numPer;
            apellido1 = ap1;
            apellido2 = ap2;
            nombre = nom;
            numAsignaturas = numAsig;
            notasFinales = notasF;
            notaMediaFinal = nmf;
        }

        public int obtenerNp(){
            return númeroPersonal;
        }

        public String toString(){
            String resul;
            resul = "NP: " + númeroPersonal + "\n" +
                    " Nombre: " + nombre +
                    " Apellido1: " + apellido1 +
                    " Apellido2: " + apellido2 + "\n" +
                    " Nota Media Final: " + notaMediaFinal +"\n";
            return resul;
        }

        public double obtenerNotaMediaFinal(){
            return notaMediaFinal;
        }
    }
```

Añada a la clase un método, de nombre asignarNotas, que reciba por parámetro una matriz de valores reales con las notas que el alumno ha obtenido en dos evaluaciones realizadas. La primera dimensión de la matriz corresponderá al número de evaluaciones y, por tanto, sólo podrá contener 2 posiciones. La segunda dimensión corresponde a las calificaciones obtenidas por el alumno en cada una de las asignaturas. El método calculará la nota final para cada una de las asignaturas teniendo en cuenta que la nota de la primera evaluación tiene un peso del 60% y la de la segunda evaluación el 40% restante. El método también determinará la nota media final del alumno.

Planteamiento: Tras comprobar que el array bidimensional recibido no es una referencia nula y que sus dimensiones se corresponden con 2 posiciones para la primera dimensión y tantas como el número de asignaturas del alumno para la segunda, el método obtiene la nota final para cada asignatura accediendo a las correspondientes posiciones de la matriz. Estas notas finales quedan almacenadas en el array `notasFinales`. Finalmente el método calcula la nota media final del alumno haciendo la media entre las calificaciones finales anteriores.

Parámetros: El método recibe como parámetro una matriz de valores reales correspondientes a las notas del alumno en 2 evaluaciones.

Valor de retorno: El método no necesita devolver ningún valor de retorno por lo que se utiliza la cláusula `void`.

Solución:

```
public void asignarNotas(double[][] notas2Eval){
    if(notas2Eval == null ||
       notas2Eval.length !=2 ||
       notas2Eval[0].length != numAsignaturas ||
       notas2Eval[1].length != numAsignaturas) {
         throw new IllegalArgumentException("Parametros no válidos.");
    }
    for(int asig = 0; asig < numAsignaturas; asig++){ ←
        notasFinales[asig] = notas2Eval[0][asig]*0.6+
                             notas2Eval[1][asig]*0.4;
    }
    for(double nota: notasFinales){ ←
        notaMediaFinal += nota;
    }
    notaMediaFinal /= numAsignaturas;
}
```

Se calcula la nota final de cada asignatura.

Se calcula la nota media final.

Ejercicio 6.35:

Añada a la clase un método, de nombre pasaDeCurso, que indique si el alumno podrá acceder al siguiente curso. Se considera que un alumno puede pasar de curso si su nota media final es mayor o igual de 5.0 y tiene menos de 3 asignaturas suspensas.

Planteamiento: Para determinar el número de asignaturas suspensas, se realiza una ordenación de las calificaciones finales del alumno. Asumiendo que el alumno siempre cursa más de tres asignaturas, se comprueba el valor de la tercera posición del array (índice 2). Si el valor de esta posición es inferior a 5.0, y por estar el array ordenado, significa que el alumno no ha logrado superar tres de sus materias. Para comprobar que la nota media final es superior a 5.0, basta llamar al método que la calcula y hacer la correspondiente comparación.

Parámetros: El método recibe como parámetro un array de cadenas de caracteres u objetos de la clase `String`.

Valor de retorno: El método devuelve la cadena resultante de concatenar todas las cadenas referenciadas desde las posiciones del array recibido, es decir, un objeto de la clase `String`.

Solución:

```
public boolean pasaDeCurso(){
    Arrays.sort(notasFinales); ←————— Se ordena el array de notas.
    if (notasFinales[2]>=5 && obtenerNotaMediaFinal()>=5.0)
        return true;
    else
        return false;
}
```

COLECCIONES

Ejercicio 6.36:

Escriba una clase, de nombre PilaPalabras, para gestionar una estructura de pila que permita apilar y desapilar objetos de la clase String. La clase implementará el método apilarPalabra para poner una palabra en la cima de la pila, el método desapilarPalabra para quitar el elemento de la cima de la pila devolviéndolo y el método obtenerPalabraCima para obtener la palabra situada en la cima de la pila sin quitarla de ella. También se implementará el método pilaPalabrasVacia para determinar si la pila está o no vacía. Los métodos deben implementarse utilizando la clase LinkedList.

Planteamiento: El constructor de la clase crea un objeto `LinkedList<String>` para simular el funcionamiento de una pila. Esto se consigue convirtiendo la operación de apilar en una inserción por el final de la lista y la operación de desapilar en la extracción del último elemento de la lista. Los métodos `add()`, `getLast()` y `removeLast()` de la clase `LinkedList` hacen que las operaciones anteriores resulten muy sencillas.

Solución:

```
public class PilaPalabras {
    LinkedList<String> linkList;

    public PilaPalabras() {
        linkList = new LinkedList<String>(); ←————— Se crea el objeto LinkedList
    }                                               para objetos de la clase String.
    public void apilarPalabra(String palabra){
      linkList.add(palabra);
    }
    public String obtenerPalabraCima(){
        return(linkList.getLast());
    }
    public String desApilarPalabra(){ ←————— Se desapila devolviendo el objeto
      return linkList.removeLast();             situado en la cima de la pila.
    }
    public boolean pilaPalabrasVacia(){
        return linkList.isEmpty();
    }
}
```

Ejercicio 6.37:

Escriba un programa que, utilizando la clase PilaPalabras del ejercicio anterior, introduzca tres cadenas de caracteres en la pila y las desapile mostrándolas por pantalla.

Planteamiento: Se crean tres cadenas de caracteres de forma estática y se introducen en la pila. Se utiliza un bucle while para extraer elementos de la pila mientras ésta no se quede vacía.

Solución:
```
public class PruebaPila{
   public static void main(String args[]){
      String cadA = "A";
      String cadB = "B";
      String cadC = "C";
      PilaPalabras p = new PilaPalabras();
      p.apilarPalabra(cadA);
      p.apilarPalabra(cadB);
      p.apilarPalabra(cadC);
      while (!p.pilaPalabrasVacia()){
         System.out.println(p.desApilarPalabra());
      }
   }
}
```

Ejercicio 6.38:

Escriba un método, de nombre obtenerArrCad5VocalesAL, que reciba por parámetro un array de cadenas y devuelva un array con las que contengan las 5 vocales. Deberá utilizarse como elemento de almacenamiento temporal un objeto de la clase ArrayList<String>. Para la consideración de un carácter como vocal no se tendrá en cuenta si está en mayúsculas o en minúsculas.

Planteamiento: El método comienza con la comprobación de que el array recibido por parámetro no se trata de una referencia nula. A continuación, se crea un objeto de la clase ArrayList<String> que almacenará las cadenas con 5 vocales. Mediante un bucle se recorren las posiciones del array recibido, para añadir al objeto ArrayList<String> las cadenas que contengan todas las vocales. Esta operación se realiza mediante el método add. Finalmente, se utilizan las facilidades que ofrece la clase ArrayList<String>, usando un método que devuelve las cadenas en un array, que finalmente se devuelve.

Parámetros: El método recibe como parámetro un array de cadenas de caracteres u objetos de la clase String.

Valor de retorno: El método devuelve un array con las cadenas presentes en el array recibido que contienen las 5 vocales, es decir, un objeto de la clase String[].

Solución:
```
public String[] obtenerArrCad5VocalesAL(String[] arrCad){
    if(arrCad == null)
        throw new IllegalArgumentException("Parametro no válido.");
    ArrayList<String> arrListCad5Vocales = new ArrayList<String>(); <-- Se construye el objeto ArrayList para objetos String.
    for(String cad : arrCad){
       if (cad != null){
           String cadAux = new String (cad.toUpperCase());
           if (cadAux.indexOf('A') != -1 &&
               cadAux.indexOf('E') != -1 &&
               cadAux.indexOf('I') != -1 &&
               cadAux.indexOf('O') != -1 &&
               cadAux.indexOf('U') != -1){
```

```
                    arrListCad5Vocales.add(i);
                }
            }
        }
        if(arrListCad5Vocales.size() != 0){
            return ((String[] )arrListCad5Vocales.toArray(new String[arrListCad5Vocales.size()]));
        }else{
            return null;
        }
    }
```

Se añaden las cadenas con todas las vocales.

Mediante este método se obtiene el contenido de la lista en forma de array. Hay que pasar un array con capacidad suficiente para almacenar los elementos de la lista.

Aviso: Para poder hacer uso de la clase `ArrayList<String>` hay que importarla del paquete `java.util`.

Ejercicio 6.39:

Escriba un método, de nombre obtenerArrListCad5Vocales, que reciba por parámetro un array de cadenas y devuelva un objeto ArrayList<String> con las que contengan las 5 vocales.

Planteamiento: El método presentado resuelve el ejercicio de forma que la complejidad de la solución se ve incrementada, pero sirve para ilustrar elementos alternativos vinculados con las colecciones de objetos. Así, el planteamiento de la solución pasa por construir un objeto `ArrayList<String>` a partir del array recibido por parámetro. Se obtiene, a continuación, un elemento de iteración (`LisIterator`) que se utilizará para eliminar de la lista aquellas cadenas que no contengan todas las vocales. Finalmente, se devuelve la lista resultante.

Parámetros: El método recibe como parámetro un array de cadenas de caracteres.

Valor de retorno: Devuelve un objeto `ArrayList<String>` que almacena las cadenas que contienen las 5 vocales.

Solución:

```
public ArrayList<String> obtenerArrListCad5Vocales(String[] arrCad){
    if(arrCad == null)
        throw new IllegalArgumentException("Parametro no válido.");
    ArrayList<String> arrListCad5Vocales = new ArrayList<String>();
    arrListCad5Vocales.addAll(((List<String>)Arrays.asList(arrCad)));
    ListIterator<String> listaIt;
    listaIt = arrListCad5Vocales.listIterator();
    while(listaIt.hasNext()){
       String cadena;
       if ( (cadena = listaIt.next()) != null){
           cadena = cadena.toUpperCase();
           if (cadena.indexOf('A') == -1 ||
               cadena.indexOf('E') == -1 ||
               cadena.indexOf('I') == -1 ||
               cadena.indexOf('O') == -1 ||
               cadena.indexOf('U') == -1){
                   listaIt.remove();            }
       }else{
           listaIt.remove();
       }
    }
    return arrListCad5Vocales;
}
```

Se crea el objeto ArrayList para los objetos String.

El método addAll() añade a la lista todos los objetos String del array.

Las referencias nulas de la lista se pueden eliminar directamente.

Aviso: Es necesario importar las clases `ArrayList`, `List` y `ListIterator` del paquete `java.util` con las sentencias:

```
import java.util.ArrayList;
import java.util.List;
import java.util.ListIterator;
```

Ejercicio 6.40:

Añada a la clase PruebaMetodosArrCad los 3 métodos anteriores como métodos estáticos e incluya en el método main() de la clase las sentencias necesarias para comprobar su correcto funcionamiento.

Planteamiento: Tras añadir los métodos como estáticos, se declara e inicializa en el método `main()` de la clase un array con 5 cadenas de caracteres. A continuación, se declara una referencia a un array de enteros y se le asigna el valor devuelto por el método `obtenerLongCadenas()` al que se especifica como argumento el array de cadenas de caracteres creado.

Solución:

```
import java.util.ArrayList;
import java.util.ListIterator;
import java.util.Arrays;
import java.util.List;
public class PruebaMetodosArrCad {
   public static int[] obtenerLongCadenas(String[] arrCad){ }
   public static String[] obtenerArrCad5VocalesAL(String[] arrCad){ }
   public static ArrayList<String> obtenerArrListCad5Vocales(String[] arrCad){ }
   public static void main(String args[]){
      String[] arc2 = { null,
                        new String("aEiou"),
                        new String("ccc"),
                        new String("pUmilotAAAe") };
      String[] c5v = PruebaMetodosArrCad.obtenerArrCad5Vocales(arc2);
      System.out.println("Cad. 5 vocales: " + Arrays.toString(c5v));
      ArrayList<String> al5v;
      al5v = PruebaMetodosArrCad.obtenerArrListCad5Vocales(arc2);
      System.out.println("Cad. 5 vocales: " + al5v);
   }
}
```

Se crea un array con una referencia nula y dos que contienen las 5 vocales distinta forma.

Se puede mostrar directamente el array con ayuda del método *toString()* de la clase *Arrays*.

Ejercicio 6.41:

Escriba un método, de nombre obtenerArrListSinCadenasRepetidas, que reciba por parámetro un objeto ArrayList<String> con cadenas de caracteres y elimine de él las sucesivas repeticiones de cada cadena.

Planteamiento: El método comprueba que el objeto recibido no constituye una referencia nula. Se utiliza un bucle para recorrer las posiciones de la lista. Para cada cadena se comprueba si su primera ocurrencia coincide con la última, es decir, que sólo figura una vez en la lista. En caso contrario, se elimina la ocurrencia actual y se continúa avanzando por los elementos de la lista. El resultado final será una lista con una sola ocurrencia de cada una de sus cadenas, es decir, una lista con todas sus cadenas distintas.

Parámetros: El método recibe como parámetro un objeto `ArrayList<String>` del que se eliminarán las sucesivas repeticiones de cada cadena.

Valor de retorno: El método no necesita devolver ningún valor de retorno por lo que se utiliza la cláusula `void`.

Solución:

```
public void obtenerArrListSinCadenasRepetidas(ArrayList<String> arrList){
    if(arrList == null)
        throw new IllegalArgumentException("Parámetro no válido.");
    for(int i = 0; i < arrList.size(); i++){
        if(arrList.lastIndexOf(arrList.get(i)) != i){ 
            arrList.remove(i);
            i--;
        }
    }
}
```

Si la ocurrencia de la cadena no es la última en la lista, la cadena se elimina.

Hay que decrementar la variable *i* al borrar pues se desplazan cubriendo el hueco.

Aviso: Para poder hacer uso de la clase ArrayList<String> hay que importarla del paquete java.util. También se podría haber realizado utilizando una clase derivada de Set, donde no puede haber objetos repetidos.

Ejercicio 6.42:

Escriba un programa que cree un conjunto ordenado según un árbol (TreeSet) utilizando la clase Alumno del Ejercicio 6.34. A continuación se crearán varios objetos de la clase Alumno, asignando a la mitad de ellos las calificaciones para obtener una nota final media de 10.0, y a otros, las necesarias para obtener una nota media final de 4.0. Se añadirán todos los alumnos al objeto TreeSet. Finalmente se obtendrán todos los alumnos aprobados (nota media final mayor o igual que 5.0) y todos los suspensos (nota media final menor que 5.0).

Planteamiento: En primer lugar es necesario realizar algunas modificaciones en la clase Alumno para que permita establecer un orden natural entre sus instancias. En primer lugar, la clase implementa el interfaz Comparable, lo que obliga a implementar el método compareTo(). Este método permite establecer una relación de orden entre dos objetos de la clase Alumno, indicando como anterior aquél con menor nota media final y como posterior el de mayor nota. En caso de que las notas resulten idénticas, el método indicará como anterior el alumno con menor número personal. Es importante tener en cuenta que esta implementación del método compareTo() nunca determinará que dos instancias de la clase Alumno son iguales, ya que al menos diferirán en el citado número personal. Se opta por esta implementación con el objetivo de garantizar que todos los alumnos se insertan en el conjunto TreeSet, puesto que si dos alumnos tuvieran la misma nota, el método los consideraría iguales y el segundo no se añadiría al conjunto. Por otro lado, el método así implementado hará posible que los alumnos queden ordenados por su nota media final y, en caso de notas iguales, por su número personal. Esta situación resulta de gran utilidad para el propósito del ejercicio.

```
public class Alumno implements Comparable<Alumno>{
    int númeroPersonal;
    String apellido1, apellido2, nombre;
    int numAsignaturas;
    double[] notasFinales;
    double notaMediaFinal;
    public Alumno(int npp, String ap1, String ap2, String nom, int numAsig){. . .}
    public Alumno(int npp, String ap1, String ap2, String nom, int numAsig,
                double[] notasF, double nmf) {. . . }
    public int obtenerNp(){. . .}
    public String toString(){. . .}
    public void asignarNotas(. . .}
    public double obtenerNotaMediaFinal(){. . .}
    public boolean pasaDeCurso(){. . .}
    public int compareTo(Alumno a1, Alumno a2){
```

```
        if ((int)(a1.obtenerNotaMediaFinal() * 100 -
                a2.obtenerNotaMediaFinal() * 100) == 0){
            return a1.obtenerNp() - a2.obtenerNp();
        }
        return (int)(a1.obtenerNotaMediaFinal() - a2.obtenerNotaMediaFinal());
    }
}
```

Se multiplica la nota por 100 para que dos notas con igual parte entera pero distinta decimal no resulten iguales.

Planteamiento: Ahora haremos el programa principal. En primer lugar se crea un objeto TreeSet. A continuación, se crean 2 matrices de notas proporcionando simultáneamente los valores de sus posiciones. Se procede a crear 8 objetos de la clase Alumno. A todos se les asignan cadenas representativas para su nombre y apellidos y un número de asignaturas igual a 5. A la mitad de los alumnos se les asignan buenas notas y a la otra mitad malas notas. A los dos últimos alumnos se les añade directamente la nota final de 5.0. Después, se añaden los objetos al objeto TreeSet. Finalmente, se utilizan los métodos headSet() y tailSet() para obtener los alumnos aprobados y suspensos. A estos métodos se les suministra un alumno de referencia cuyos atributos tiene valores por defecto salvo su número personal que deber ser 0 y su nota media final que interesa inicializar al valor 5.0 para que la comparación resulte correcta.

```
public class EjTreeSet {
    public static void main(String args[]){
        TreeSet<Alumno> treeSetAlum;
        treeSetAlum = new TreeSet<Alumno>();
        Alumno a1 = new Alumno(1, "Alum1", "Alum1", "Alum1", 5);
        Alumno a2 = new Alumno(2, "Alum2", "Alum2", "Alum2", 5);
        Alumno a3 = new Alumno(3, "Alum3", "Alum3", "Alum3", 5);
        Alumno a4 = new Alumno(4, "Alum4", "Alum4", "Alum4", 5);
        Alumno a5 = new Alumno(5, "Alum5", "Alum5", "Alum5", 5);
        Alumno a6 = new Alumno(6, "Alum6", "Alum6", "Alum6", 5);
        Alumno a7 = new Alumno(7, "Alum7", "Alum7", "Alum7", 5, null, 5.0);
        Alumno a8 = new Alumno(8, "Alum8", "Alum8", "Alum8", 5, null, 5.0);
        double[][] buenasNotas ={{10.0, 10.0, 10.0, 10.0, 10.0},
                                 {10.0, 10.0, 10.0, 10.0, 10.0}};
        double[][] malsanotas ={{4.0, 4.0, 4.0, 4.0, 4.0},
                                {4.0, 4.0, 4.0, 4.0, 4.0}};
        a1.asignarNotas(buenasNotas);
        a2.asignarNotas(malasNotas);
        a3.asignarNotas(buenasNotas);
        a4.asignarNotas(malasNotas);
        a5.asignarNotas(buenasNotas);
        a6.asignarNotas(malasNotas);
        treeSetAlum.add(a1);
        treeSetAlum.add(a2);
        treeSetAlum.add(a3);
        treeSetAlum.add(a4);
        treeSetAlum.add(a5);
        treeSetAlum.add(a6);
        treeSetAlum.add(a7);
        treeSetAlum.add(a8);

        SortedSet<Alumno> aprobados;
        Aprobados = treeSetAlum.tailSet(new Alumno(0, null, null, null, 0, null, 5.0));
```

Se crea el objeto TreeSet.

Se asignan las notas a los alumnos.

Se añaden los alumnos al objeto TreeSet.

Se obtienen como aprobados aquellos Alumnos posteriores según el orden natural al suministrado como argumento.

```
        SortedSet<Alumno> suspensos;
        Suspensos = treeSetAlum.headSet(new Alumno(0, null, null, null, 0, null, 5.0));

        System.out.println("Alumnos aprobados: ");
        System.out.println(aprobados);
        System.out.println("Alumnos suspensos: ");
        System.out.println(suspensos);
    }
}
```

Comentario: Al obtener los alumnos aprobados es importante proporcionar un objeto con valor de `númeroPersonal` igual a 0. De esta forma cualquier objeto correspondiente a un alumno con nota media final igual a 5.0 será devuelto como superior por tener mayor valor de `númeroPersonal`. (Se supondrá este atributo como mayor estricto que 0 para todos los objetos de la clase Alumno.)

Ejercicio 6.43:

Utilizando la clase Alumno del Ejercicio 6.34, escriba una nueva clase, de nombre VectorAlumnos, teniendo en cuenta las siguientes especificaciones

a) La clase tendrá como único atributo un objeto de la clase Vector<Alumno> de nombre alumnos.
b) La clase tendrá un constructor que reciba por parámetro un array de alumnos para obtener a partir de él un objeto de la clase Vector<Alumno> que almacene los objetos alumnos del array y que se asignen al atributo alumnos.
c) La clase dispondrá de un método, de nombre obtenerAlumnosNMFAprobado, que a partir del vector almacenado en el atributo alumnos devolverá un vector únicamente con los alumnos aprobados, es decir, con nota media final superior a 5.0 .

Planteamiento: En primer lugar, se declara el atributo especificado para la clase: un objeto de la clase `Vector<Alumno>` de nombre alumnos. A continuación se escribe un método constructor para la clase que recibe un array de objetos de la clase Alumno. El constructor obtiene una lista de objetos `Alumno` utilizando el método `asList()` de la clase `Arrays` y la suministra al constructor de la clase `Vector<Alumno>` para instanciar un objeto vector con los elementos presentes en la lista.

El método `obtenerAlumnosNMFAprobado()` debe obtener a partir del atributo alumnos un objeto `Vector<Alumnos>` que almacene únicamente los alumnos aprobados. Para ello el método comienza ordenando el vector alumnos mediante el método `sort()` de la clase `Collections`. Para llevar a cabo la comparación se instancia un objeto de la clase `Comparator`. Esta clase implementa la interfaz `Comparator<Alumno>` y define el método `compare()` para indicar cómo realizar la ordenación. Este método establece que dos alumnos quedan ordenados en función de su nota media final y en caso de que ésta coincida por su número personal. Una vez ordenado, se localiza en el vector el primer alumno aprobado. Se utiliza el índice de este alumno en el vector ordenado para obtener un subvector que vaya desde él hasta el final del vector. Por estar ordenado en función de la nota, este subvector corresponderá a los alumnos aprobados. Finalmente, se devuelve el subvector obtenido.

Solución:

```
class Comparador implements Comparator<Alumno>{ <——
    public int compare(Alumno a1, Alumno a2){
      if ((int)(a1.obtenerNotaMediaFinal() * 100 -
                a2.obtenerNotaMediaFinal() * 100) == 0){
          return a1.obtenerNp() - a2.obtenerNp();
      }
      return (int)(a1.obtenerNotaMediaFinal() - a2.obtenerNotaMediaFinal());
```

Clase del objeto utilizado como comparador de los alumnos.

```
    }
}

public class VectorAlumnos {
    Vector<Alumno> alumnos;

    public VectorAlumnos(Alumno[] alums) {
      alumnos = new Vector<Alumno>((List<Alumno>)(Arrays.asList(alums)));
    }

    public Vector<Alumno> obtenerAlumnosNMFAprobado(){
        Collections.sort(alumnos, new Comparador());
        ListIterator<Alumno> vecIt = (ListIterator<Alumno>)alumnos.listIterator();
        boolean encontrado = false;
        Alumno primerAprobado = null;
        while (vecIt.hasNext() && !encontrado){
            primerAprobado = vecIt.next();
            if (primerAprobado.obtenerNotaMediaFinal()>=5.0){
                encontrado = true;
            }
        }
        Vector<Alumno> aprobados =
            new Vector<Alumno>(alumnos.subList(alumnos.indexOf(primerAprobado),
                                                alumnos.size()));
        return aprobados;
    }
```

Construcción del objeto Vector<Alumno> a partir del array de alumnos recibido por parámetro.

Obtención del subvector con los alumnos aprobados.

Comentario: En realidad no sería necesario utilizar el objeto comparador porque define el mismo orden que el método compareTo() de la clase Alumno, de forma que la sentencia

```
Collections.sort(alumnos);
```

daría lugar a la misma ordenación que la mostrada en el ejemplo. Se ha incluido para ilustrar la forma de ordenación de los elementos de una colección en un orden distinto al orden natural, que establecería el método compareTo() de la clase de dichos elementos.

Ejercicio 6.44:

Escriba un programa para verificar el correcto funcionamiento de los métodos escritos como parte de la clase VectorAlumnos. El programa creará una serie de objetos de la clase Alumno a los que asignará buenas y malas notas. Estos objetos se incluirán en un array del que se creará un objeto de la clase VectorAlumnos. Una vez creado el objeto se obtendrá el vector correspondiente a los alumnos aprobados, mostrando sus datos y calificaciones por pantalla.

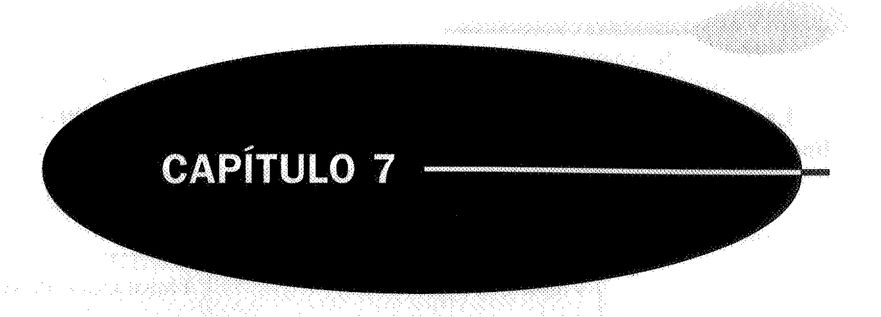
Planteamiento: Se procede a la creación de los objetos de la clase Alumno para asignarles después las calificaciones y añadirlos a un array. Se crea un objeto de la clase VectorAlumnos suministrando el array anterior al constructor de la clase. Se obtiene en un objeto Vector<Alumno> el vector correspondiente a los alumnos aprobados. Finalmente se procede a mostrar los datos de tales alumnos por pantalla.

Solución:

```
public static void main(String args[]){
    Alumno a1 = new Alumno(1, "Alum1", "Alum1", "Alum1", 5);
```

```
        Alumno a2 = new Alumno(2, "Alum2", "Alum2", "Alum2", 5);
        Alumno a3 = new Alumno(3, "Alum3", "Alum3", "Alum3", 5);
        Alumno a4 = new Alumno(4, "Alum4", "Alum4", "Alum4", 5);
        Alumno a5 = new Alumno(5, "Alum5", "Alum5", "Alum5", 5);
        Alumno a6 = new Alumno(6, "Alum6", "Alum6", "Alum6", 5);
        Alumno a7 = new Alumno(7, "Alum7", "Alum7", "Alum7", 5, null, 5.0);
        Alumno a8 = new Alumno(8, "Alum8", "Alum8", "Alum8", 5, null, 5.0);
        double[][] buenasNotas = {{10.0, 10.0, 10.0, 10.0, 10.0},
                                  {10.0, 10.0, 10.0, 10.0, 10.0}};
        double[][] malasNotas = {{4.0, 4.0, 4.0, 4.0, 4.0},
                                 {4.0, 4.0, 4.0, 4.0, 4.0}};
        a1.asignarNotas(buenasNotas);
        a2.asignarNotas(malasNotas);
        a3.asignarNotas(buenasNotas);
        a4.asignarNotas(malasNotas);
        a5.asignarNotas(buenasNotas);
        a6.asignarNotas(malasNotas);
        Alumno[] alumnos = {a1, a2, a3, a4, a5, a6, a7, a8};
        VectorAlumnos va = new VectorAlumnos(alumnos);
        Vector<Alumno> aprobados = va.obtenerAlumnosNMFAprobado();
        System.out.println(aprobados);
    }
```

Creación del objeto VectorAlumno con los alumnos almacenados en el array.



Entrada y salida

7.1 CONCEPTO DE FLUJO EN JAVA

La información que necesita un programa para su función se obtiene mediante una *entrada* de datos de una fuente que puede ser de tipos muy variados: desde el teclado, desde un archivo, desde una comunicación en red, desde un objeto en Internet, etc. Así mismo, el tipo de datos que se lee puede ser de muy diversas características: texto, imágenes, sonidos, etc.

Cuando el programa genera los resultados como *salida* de la ejecución puede hacerlo de muy diversas maneras: en un archivo, en la pantalla, en una impresora, etc., y la forma como genera este resultado también puede ser de muy diferente tipo: texto, binario, imágenes, etc.

En Java la entrada (lectura) de los datos se realiza mediante un flujo de entrada. La salida (escritura) de datos se realiza mediante un flujo de salida.

El esquema para trabajar con los flujos de datos tanto de entrada como de salida es el que se muestra el la Tabla 7.1.

Tabla 7.1. Esquema para trabajar con los flujos de datos de entrada/salida.

Entrada de datos (leer datos)	Salida de datos (escribir datos)
1. Se crea un objeto flujo de datos de lectura	1. Se crea un objeto flujo de datos de escritura
2. Se leen datos de él con los métodos apropiados	2. Se escriben datos utilizando los métodos apropiados del objeto flujo
3. Se cierra el flujo de datos	3. Se cierra el flujo de datos

Ambas operaciones se pueden entremezclar creando objetos flujo para leer y para escribir, leyendo de uno y escribiendo de otro para, finalmente, cerrar tanto el flujo de lectura como el de escritura.

7.2 TIPOS DE FLUJOS

Existen dos tipos de flujos definidos en Java: unos que trabajan con bytes y otros que trabajan con caracteres. Así mismo existen clases conversoras que permiten obtener un flujo de bytes a partir de uno de caracteres y viceversa, tanto para lectura como para escritura.

Las clases más importantes a tener en cuenta son las que se muestran en la Tabla 7.2, donde el sangrado de las líneas indica la herencia, es decir, `DataInputStream` hereda de `FilterInputStream` que, a su vez, hereda de `InputStream`.

Tabla 7.2. Las clases más importantes a tener en cuenta.

	Flujos con bytes	Flujos con caracteres
Entrada de datos	InputStream	Reader
	ByteArrayInputStream	BufferedReader
	FileInputStream	LineNumberReader
	FilterInputStream	CharArrayReader
	BufferedInputStream	FilterReader
	DataInputStream	PushbackReader
	LineNumberInputStream	InputStreamReader
	PushbackInputStream	FileReader
	ObjectInputStream	PipedReader
	PipedInputStream	StringReader
	SequenceInputStream	
	StringBufferInputStream	
Salida de datos	OutputStream	Writer
	ByteArrayOutputStream	BufferedWriter
	FileOutputStream	CharArrayWriter
	FilterOutputStream	FilterWriter
	BufferedOutputStream	OutputStreamWriter
	DataOutputStream	FileWriter
	PrintStream	PipedWriter
	ObjectOutputStream	PrintWriter
	PipedOutputStream	StringWriter

Además de las clases anteriores existe una clase especial en el paquete `java.io` de nombre `RandomAccessFile` para el acceso aleatorio a archivos. Permite utilizar los archivos en modo lectura y escritura simultáneamente o acceder a datos de forma aleatoria indicando la posición en la que se quiere operar.

7.3 LEER Y ESCRIBIR EN UN ARCHIVO

Un archivo se encuentra en un disco. Desde el punto de vista de lectura y escritura un archivo se trata de una secuencia continua de datos, ya sean bytes o caracteres. En la Figura 7.1 se puede observar una representación de un archivo de caracteres. Se puede ver cómo se numeran las casillas del archivo y un cursor que apunta al lugar de la siguiente operación.

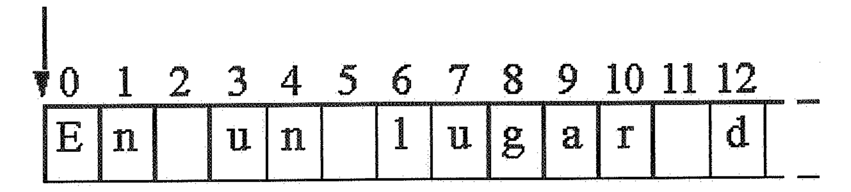


Figura 7.1. Representación de un archivo de caracteres.

La forma básica de utilizar un archivo para entrada y para salida de datos, para leer del archivo y para escribir respectivamente, se resume en la Tabla 7.3.

Tabla 7.3. Forma básica de utilizar un archivo para entrada y salida de datos.

Lectura de un archivo	Escritura de un archivo
Abrir el flujo del archivo Mientras queden datos leer el siguiente dato Cerrar el flujo del archivo	Abrir el flujo del archivo Mientras haya datos por escribir escribir en el archivo Cerrar el flujo del archivo

En el siguiente ejemplo se presenta un programa que abre un archivo llamado "prueba.txt", escribe en el archivo caracteres desde el carácter 'a' hasta el carácter 'z', carácter a carácter, y lo cierra.

A continuación, para comprobar si se ha escrito bien, abre el archivo para leer, lee todo lo que contiene, carácter a carácter, y escribe por pantalla los caracteres leídos. Para terminar cierra el archivo.

```
import java.io.*;

public class PruebaArchivos {
    public static void main(String arg[]) {
        String nombreArchivo = "prueba.txt";
        FileWriter escribir;
        try {
            escribir = new FileWriter(nombreArchivo);
            for (char c = 'a'; c <= 'z'; c++) {
                escribir.write(c);
            }
            escribir.close();
        } catch (IOException e) {
            System.out.println("Imposible abrir el archivo para escribir.");
        }
        FileReader leer;
        int c;
        try {
            leer = new FileReader(nombreArchivo);
            c = leer.read();
            while (c != -1) {
                System.out.print((char)c);
                c = leer.read();
            }
            leer.close();
        } catch (IOException e) {
            System.out.println("Imposible abrir el archivo para leer.");
        }
    }
}
```

Para escribir en el archivo, se abre el flujo del archivo con la sentencia:

```
FileWriter archivo = new FileWriter("prueba.txt");
```

Si el archivo no existe, lo crea. Si el archivo ya existe, elimina su contenido y empezará a escribir como si estuviese vacío. Si no puede abrir el archivo para escribir, genera una excepción.

Para leer del archivo se abre el flujo utilizando la sentencia:

```
FileReader archivo = new FileReader("prueba.txt");
```

Esta sentencia intenta abrir el archivo especificado. Si existe y se puede abrir para lectura, lo abre. Si existe y no lo puede abrir para lectura o si no existe, genera una excepción.

En ambos casos una vez se ha terminado de escribir o de leer, se cierra el flujo utilizando el método `close()` sobre el objeto flujo.

Como curiosidad reseñar la forma en que se hace la lectura de los caracteres. Cuando se van leyendo caracteres del flujo `FileReader` se utiliza el método `read()` del flujo. Pero este método devuelve un valor del tipo `int`, no un valor del tipo `char`. Al devolver un `int` se indica que devuelve un valor de carácter (entre 0 y 65535, ya que los `char` se codifican con 16 bits), o bien devuelve un valor sin sentido para los caracteres, en este caso el valor -1, para indicar que ya no hay más caracteres. Este valor especial es el que se utiliza en el bucle para saber que ya se ha acabado de leer.

7.4 FILTROS

Se denominan filtros a las clases que representan un flujo de datos, ya sean de lectura o de escritura pero cuyo origen o destino de los datos es otro flujo. Estos flujos (filtros) que se conectan a otros flujos que ya existen para, leyendo a través de los mismos transformar los datos, permiten proporcionar métodos de lectura o escritura más apropiados al programador.

Por ejemplo, el filtro de la clase `BufferedReader` se puede conectar a un objeto flujo de la clase `FileReader`, como el que se ha usado en el ejemplo anterior. La clase `BufferedReader` dispone de un método para leer una línea completa de una sola vez. Este método se llama `readLine()`. Cada vez que se llama al método `readLine()` de la clase `BufferedReader` este método, por la forma en que están enlazados, lo que hace es llamar las veces que precise al método `read()` de la clase `FileReader` hasta conseguir una línea completa. Cuando la tiene devuelve la línea completa al programa que llamó a `readLine()`.

El siguiente ejemplo muestra cómo se puede utilizar el filtro `BufferedReader` para escribir en la pantalla el contenido de un archivo, que en este caso se trata del propio programa en Java.

```
import java.io.*;

public class LeerArchivo {

    public static void main(String arg[]) {
        String nombreArchivo="LeerArchivo.java";

        FileReader archivo;
        BufferedReader filtro;
        String línea;
        try {
            archivo = new FileReader(nombreArchivo);
            filtro  = new BufferedReader(archivo);
            línea   = filtro.readLine();
            while (línea != null) {
                System.out.println(línea);
                línea = filtro.readLine();
            }
            filtro.close();
        } catch (IOException e) {
            System.out.println("Imposible abrir el archivo para leer.");
```

```
        }
      }
    }
```

En el ejemplo anterior se puede observar cómo se realiza la conexión entre el flujo `FileReader` y el filtro `BufferedReader`. De hecho, puede observar que la conexión se establece en las siguientes dos líneas:

```
FileReader archivo = new FileReader("prueba.txt");
BufferedReader filtro = new BufferedReader(archivo);
```

En la primera se abre el flujo, igual que en el primer ejemplo, y en la segunda línea, en el constructor del objeto, se indica cuál es el flujo de donde se leerán los datos.

Se pueden encadenar tantos filtros como sea necesario.

7.5 ENTRADA DESDE TECLADO

Para la lectura de datos del teclado, en Java se dispone de la clase Scanner que ya se ha comentado en el Capítulo 1 y donde puede encontrar ejercicios que usan Scanner para solicitar datos al usuario. De todas formas la potencia del uso de los flujos permite al programador utilizarlos de muchas formas. Se va a mostrar cómo utilizar flujos para leer datos de teclado.

Java proporciona un flujo para la entrada predeterminada llamado `System.in` que el sistema abre al empezar la ejecución del programa. Este flujo lee, por defecto, del teclado. Así mismo se dispone del flujo `System.out` para la salida predeterminada. Este flujo escribe, por defecto, en la pantalla, en la consola de Java. Ambos flujos predeterminados son flujos de bytes.

En el siguiente ejemplo se van a leer distintos tipos de datos del teclado como líneas completas que introduzca el usuario y, posteriormente, convertirlas al tipo de dato que nos interese. En la parte final del programa se ha escrito un fragmento de código que permite solicitar al usuario un número hasta que introduzca un valor correcto para el tipo de número que se está solicitando. Fíjese en cómo saber si la conversión ha sido correcta manejando las excepciones.

```
import java.io.*;

public class LeerTeclado {

    public static void main(String arg[]) throws IOException{
        InputStreamReader conversor;
        BufferedReader teclado;
        String línea;

        conversor = new InputStreamReader(System.in);
        teclado = new BufferedReader(conversor);

        System.out.print("Introduzca un byte: ");
        línea = teclado.readLine();
        byte b = Byte.parseByte(línea);
        System.out.println("El valor leído fue: " + b);

        System.out.print("Introduzca un int: ");
        línea = teclado.readLine();
        int i = Integer.parseInt(línea);
        System.out.println("El valor leído fue: " + i);
```

```
            System.out.print("Introduzca un double: ");
            línea = teclado.readLine();
            double d = Double.parseDouble(línea);
            System.out.println("El valor leido fue: " + d);

            boolean leido;
            do {
                try {
                    System.out.print("Introduzca un int: ");
                    línea = teclado.readLine();
                    i = Integer.parseInt(línea);
                    leido = true;
                } catch (NumberFormatException e) {
                    System.out.println("Número no válido. Vuelva a intentarlo.");
                    leido = false;
                }
            } while (!leido);

            System.out.println("El valor leído fue: " + i);
        }
    }
```

7.6 LA CLASE FILE

La clase `File` no sirve para leer ni para escribir en un archivo sino que permite, entre otras operaciones:

- Obtener el tamaño del archivo.
- Obtener el nombre completo, incluida la ruta.
- Cambiar el nombre.
- Eliminar el nombre.
- Saber si es un directorio o un archivo.
- Si es un directorio, obtener la lista de los archivos y directorios que contiene.
- Crear un directorio.

En el siguiente ejemplo puede ver un programa que muestra el contenido de un directorio. Si no se indica nada en la línea de comandos, muestra el contenido del directorio actual. Si se indica un directorio, muestra el contenido del directorio indicado. Si se encuentra un nombre de directorio, sólo se indica el nombre. Si se encuentra el nombre de un archivo, se muestra su tamaño y fecha.

```
import java.io.*;
import java.util.*;

public class ListarDirectorio {

    public static void main (String arg[]) {
        String directorio;
        if (arg.length > 0) {
            directorio = arg[0];
        } else {
            directorio = ".";
```

```
        }

        File actual = new File(directorio);
        System.out.print("El directorio es: ");
        try {
            System.out.println(actual.getCanonicalPath());
        } catch (IOException e) {
        }
        System.out.println("Su contenido es:");

        File[] archivos = actual.listFiles();
        for(File archivo : archivos){
            System.out.printf("%-15s",archivo.getName());
            if(archivo.isFile()){
                System.out.printf("%6d  ",archivo.length());
                System.out.printf("%1$tD %1$tT", new Date(archivo.lastModified()));
            }
            System.out.println();
        }
    }
}
```

7.7 ARCHIVOS DE ACCESO ALEATORIO

La clase `RandomAccessFile` permite abrir un archivo como de lectura, o de lectura y escritura simultáneamente. Si se utiliza para lectura del archivo (modo "r"), dispone de métodos para leer elementos de cualquier tipo primitivo: `readInt()`, `readLong()`, `readDouble()`, `readLine()`, etc. Así mismo, cuando se utiliza en el modo de lectura y escritura (modo "rw") se pueden utilizar los métodos de escritura para escribir los tipos de datos de forma similar a como se pueden leer con los métodos: `writeInt()`, `writeLong()`, `writeDouble()`, `writeBytes()`, etc.

Los métodos que resultan de interés para el acceso aleatorio son los que permiten acceder a un lugar concreto dentro del archivo y conocer el punto del mismo en el que se va a realizar la operación de lectura y/o escritura:

- `getFilePosition()`: Devuelve la posición actual donde se va a realizar la operación de lectura o escritura. Devuelve la posición, contando en bytes donde se encuentra actualmente el cursor del archivo.
- `seek()`: Sitúa la posición de la próxima operación de lectura o escritura en el byte especificado.
- `length()`: Devuelve el tamaño actual del archivo.

En el siguiente ejemplo se puede ver un programa que se utiliza para ir leyendo carácter a carácter un archivo. Lo que hace es convertir a mayúsculas todas los caracteres 'b' que encuentre. Cuando encuentra una b minúscula vuelve una posición atrás de donde la había leído y escribe el mismo carácter convertido a mayúsculas. El resultado es que el archivo de texto aparece con todas las letras b en mayúsculas.

```
import java.io.*;

public class ArchivoAleatorio {

    public static void main(String arg[]) {
        char c;
        boolean finArchivo = false;
        RandomAccessFile archivo = null;
```

```
        try {
            archivo = new RandomAccessFile("prueba.txt", "rw");
            System.out.println("El tamaño es: " + archivo.length());
            do {
                try {
                    c = (char)archivo.readByte();
                    if (c == 'b'){
                        archivo.seek(archivo.getFilePointer()-1);
                        archivo.writeByte(Character.toUpperCase(c));
                    }
                } catch (EOFException e) {
                    finArchivo = true;
                    archivo.close();
                    System.out.println("Convertidas las b a mayúsculas.");
                }
            } while (!finArchivo);
        } catch (FileNotFoundException e) {
            System.out.println("No se encontró el archivo.");
        } catch (IOException e) {
            System.out.println("Problemas con el archivo.");
        }
    }
}
```

Fíjese en la forma de detectar que se ha llegado al final del archivo. Los métodos de la clase RandomAccessFile generan una excepción, EOFException, para indicar que se ha intentado leer fuera del archivo. Por eso, para detectar que se ha terminado de leer se utiliza una variable finArchivo que vale false mientras se siga leyendo del archivo. Cuando se genere la excepción EOFException, se captura, se cierra el archivo y se pone el valor true en la variable finArchivo. Al cambiar este valor termina el bucle de lectura.

7.8 LECTURA Y ESCRITURA DE OBJETOS

Se puede utilizar la capacidad de Java para dar soporte a la serialización de objetos para poder leer y escribir objetos completos sin preocuparse de cómo están implementados por dentro. Para que una clase sea serializable debe de implementar la interfaz Serializable. Todas las clases de la API estándar de Java son serializables.

Para escribir objetos de una clase, que implementa la interfaz Serializable, se utiliza el flujo ObjectOutputStream. En la serialización de un objeto se escriben en el flujo todos los atributos que no sean ni static ni transient. Si al escribir los atributos de un objeto alguno de ellos es un objeto, se serializa a su vez dicho objeto, serializando todos sus atributos. De esta forma cuando se lee el objeto se podrá reconstruir tal y como estaba en la parte que no es static ni transient. Para escribir un objeto se utiliza el método writeObject().

Para leer un objeto escrito mediante serialización se utiliza el flujo ObjectInputStream. Una vez creado un objeto de este flujo, se puede llamar al método readObjetct() para leer un objeto del flujo. Este método devuelve un Object por lo que habrá que realizar una conversión de clase de la forma apropiada.

En el siguiente ejemplo se puede ver el uso de la serialización de objetos tanto para escribirlos como para leerlos. Se ha utilizado para ello un objeto String y un objeto de la clase Date implementando ambos la interfaz Serializable.

```
import java.util.*;
import java.io.*;

public class Serial {
```

```
    public static void main(String arg[]) {
        try {
            FileOutputStream archivo  = new FileOutputStream("prueba.dat");
            ObjectOutputStream salida = new ObjectOutputStream(archivo);
            salida.writeObject("Hoy es: ");
            salida.writeObject(new Date());
            salida.close();
        } catch (IOException e) {
            System.out.println("Problemas con el archivo.");
        }

        try {
            FileInputStream archivo2  = new FileInputStream("prueba.dat");
            ObjectInputStream entrada = new ObjectInputStream(archivo2);
            String hoy = (String)entrada.readObject();
            Date fecha = (Date)entrada.readObject();
            entrada.close();

            System.out.println(hoy + fecha);
        } catch (FileNotFoundException e) {
            System.out.println("No se pudo abrir el archivo.");
        } catch (IOException e) {
            System.out.println("Problemas con el archivo.");
        } catch (Exception e) {
            System.out.println("Error al leer un objeto.");
        }
    }
}
```

Si se desea se puede rescribir los métodos de lectura y escritura de los objetos para que la clase propia realice toda la operación de serialización.



Problemas resueltos

TIPOS DE FLUJOS

Ejercicio 7.1:

Escriba un programa, de nombre EscribeArray, que cree un array de bytes con los dígitos del 0 al 9 y, a continuación, defina sobre él un flujo de entrada para leer sus valores y mostrarlos por pantalla.

Planteamiento: El array de valores bytes se crea de forma estática indicando directamente los dígitos que va a almacenar. A continuación, se define un flujo de entrada de la clase `ByteArrayInputStream` con el array. El método `available()` permite determinar el número de bytes que pueden leerse del flujo en cada instante. De esta forma, mediante un bucle de tipo `while` se procede a leer todos los valores del flujo mediante el método `read()`. Cada lectura sobre el flujo decrementará en una unidad el número de bytes disponibles en él. Esta circunstancia se aprovecha para implementar la condición de control del bucle.

Solución:

```
class EscribeArray{
   public static void main(String args[]){
      byte[] byteArr1 = {0, 1, 2, 3, 4, 5, 6, 7, 8 , 9};
      ByteArrayInputStream flujoArrByte1 = new ByteArrayInputStream(byteArr1);
      while(flujoArrByte1.available() != 0){
            byte leido = (byte) flujoArrByte1.read();
            System.out.println(leido);
      }
      flujoArrByte1.close();
   }
}
```

Creación del flujo de entrada.

En cada iteración del bucle se lee un nuevo valor del flujo.

Ejercicio 7.2:

Escriba un programa, de nombre ValoresPares, que cree un array de bytes con los dígitos del 0 al 9 y, a continuación, defina sobre él un flujo de entrada para leer sus valores. La lectura se realizará de forma que después de leer un valor del flujo se descartará la lectura del siguiente.

Planteamiento: Se crea e inicializa directamente el array con los valores de tipo `byte`. A continuación, se crea un flujo de entrada de la clase `ByteArrayInputStream` que lee del array. Se procede a la lectura del array utilizando el método `read()` sobre el flujo creado. Para descartar un valor del flujo después de una lectura se utiliza el método `skip()` que permite saltar el número de bytes del flujo especificado por su argumento. Para el array creado, esta acción supondrá que sólo se muestren por pantalla los valores correspondientes a las posiciones pares del array.

Solución:

```
class ValoresPares {
   public static void main(String args[]){
      byte[] byteArr1 = {0, 1, 2, 3, 4, 5, 6, 7, 8 , 9};
      ByteArrayInputStream flujoArrByte1 = new ByteArrayInputStream(byteArr1);
      while(flujoArrByte1.available() != 0){
         byte leido =(byte)flujoArrByte1.read();
         System.out.println(leido);
         flujoArrByte1.skip(1);
```

Después de una lectura se salta el siguiente valor del flujo.

```
            }
            flujoArrByte1.close();
        }
    }
```

Ejercicio 7.3:

Escriba un programa, de nombre CombinarFlujos, que combine en un único flujo de entrada, dos flujos procedentes de sendos arrays de valores de tipo byte. El programa creará dos arrays dando valor a sus elementos, creará dos flujos de entrada asociados a ellos y los combinará en uno solo a partir del cual se pueden leer los elementos de los dos arrays para mostrarlos por pantalla.

Planteamiento: Tras crear e inicializar dos arrays de valores de tipo `byte`, se crean dos flujos de entrada asociados a ellos. Estos flujos se combinan en uno solo utilizando un objeto de la clase `SequenceInputStream`. Finalmente se utiliza este objeto para leer de él los valores correspondientes a los dos flujos combinados.

Solución:

```
class CombinarFlujos{
    public static void main(String args[]){
        byte[] byteArr1 = {0, 1, 2, 3, 4, 5, 6, 7, 8 , 9};
        byte[] byteArr2 = {10, 20, 30, 40, 50, 60, 80, 80, 80, 100};
        ByteArrayInputStream flujoArrByte1 = new ByteArrayInputStream(byteArr1);
        ByteArrayInputStream flujoArrByte2 = new ByteArrayInputStream(byteArr2);
        SequenceInputStream flujoSec = new SequenceInputStream(flujoArrByte1,
                                                               flujoArrByte2);
        try{
            byte b = 0;
            while( (b = (byte)flujoSec.read()) != -1){
                System.out.println(b);
            }
            flujoArrByte1.close();
            flujoArrByte2.close();
            flujoSec.close();
        }catch(IOException ioE){
            System.out.println(ioE.toString());
        }
    }
}
```

Se combinan los flujos de entrada de los dos arrays en uno solo.

El método `read()` puede lanzar excepciones de la clase `IOException` que hay que capturar.

Ejercicio 7.4:

Escriba un método, de nombre obtenerMatrizIdentidad, que cree una matriz cuadrada a partir de la dimensión que recibe por parámetro y cree una matriz identidad (la matriz deberá tener el valor 1 en todos los elementos de su diagonal principal y 0 en el resto de posiciones). La asignación de valores a la matriz se realizará a través de un flujo de escritura definiendo cada elemento de la primera dimensión (filas) para especificar los valores sobre la segunda.

Planteamiento: En primer lugar se comprueba que la dimensión recibida es adecuada para la construcción de la matriz. A continuación, se procede a la creación de la matriz especificando sólo el número de elementos en la primera dimensión. Dentro de un bucle se define un flujo de escritura sobre cada elemento de la primera dimensión que permitirá escribir los valores que quedarán almacenados a lo largo de la segunda. Para la escritura se tiene en cuenta que sólo se escribe el valor 1 cuando coincidan los índices según las dos dimensiones.

Parámetros: Valor entero correspondiente a la dimensión de la matriz identidad a crear

Valor de retorno: Matriz de valores byte correspondiente a la matriz identidad para la dimensión recibida por parámetro.

Solución:
```
public byte[][] obtenerMatrizIdentidad(int dimension){
    if(dimension <= 0){
        throw new IllegalArgumentException("Argumento incorrecto");
    }
    byte[][] matriz = new byte[dimension][];
    for(int i = 0; i < matriz.length; i++){
        ByteArrayOutputStream fila = new ByteArrayOutputStream(dimension);
        for(int j = 0; j < dimension; j++){
            fila.write(i == j ? 1 : 0);
        }
        try{
            fila.close();
        } catch(IOException ioE){}
        matriz[i] = fila.toByteArray();
    }
    return matriz;
}
```

Se crea un flujo de escritura para escribir los elementos correspondientes a cada fila.

El método toByteArray() convierte el flujo en un array que se asigna al correspondiente elemento de la matriz.

Ejercicio 7.5:

Escriba un método, de nombre mostrarMatrizConFlujos, que reciba por parámetro una matriz con valores de tipo byte para mostrarlos por pantalla. El método deberá acceder a los valores a lo largo de la segunda dimensión de la matriz mediante flujos de entrada creados sobre los elementos de la primera.

Planteamiento: El método recorre los elementos de la matriz según su primera dimensión para definir sobre cada uno de ellos un flujo de entrada que permita leer los valores correspondientes a la segunda. El método cuida que los elementos sean presentados por pantalla con apariencia matricial de filas y columnas.

Parámetros: Matriz con valores de tipo byte para mostrarla por pantalla.

Valor de retorno: El método no devuelve ningún valor de retorno, por tanto utiliza la cláusula void.

Solución:
```
public void mostrarMatrizConFlujos(byte[][] matriz) {

for(int i = 0; i < matriz.length; i++){
        ByteArrayInputStream fila = new ByteArrayInputStream(matriz[i]);
        byte elemento;
        while( (elemento = (byte)fila.read()) != -1){
            System.out.print(elemento + " ");
        }
        System.out.println();
    }
}
```

Se crea un flujo de lectura para recorrer los elementos a lo largo de la segunda dimensión de la matriz.

Ejercicio 7.6:

Escriba un método, de nombre clasificadorMinMay, que reciba por parámetro una cadena de caracteres para clasificarlos en función de que sean letras minúsculas o mayúsculas. El método deberá, así, obtener una cadena con las letras en minúscula y otra distinta con las letras en mayúscula. Las cadenas obtenidas se asignarán a las posiciones de un array que finalmente se devolverá. Tanto la lectura de los caracteres de la cadena recibida por parámetro como la escritura a las cadenas obtenidas, se realizará mediante flujos de lectura y escritura.

Planteamiento: Se crea un flujo de entrada para la lectura de los caracteres de la cadena recibida y dos flujos de escritura que se corresponderán, respectivamente, con la cadena para las letras minúsculas y con la que contendrá las letras mayúsculas. Mediante un bucle se lee cada carácter de la cadena recibida por parámetro y tras comprobar si se trata de una letra en minúscula o en mayúscula se añade, utilizando el método `append()`, al correspondiente flujo. Los flujos son completamente vaciados mediante el método `flush()` y con el método `toString()` transformados en cadenas de caracteres para ser asignadas a las posiciones del array que, finalmente, será devuelto.

Parámetros: Cadena de caracteres cuyos caracteres serán clasificados en letras minúsculas y mayúsculas. Los dígitos y resto de caracteres serán descartados de la clasificación.

Valor de retorno: Array de objetos de la clase `String`. La primera posición del array corresponderá a la cadena formada con las letras minúsculas encontradas en la cadena recibida, mientras que la segunda corresponderá a la cadena con las letras mayúsculas.

Solución:

```
public String[] ClasificadorMinMay(String cad) throws IOException{
   StringReader flujoLecCad = new StringReader(cad);
   StringWriter flujoEscMin = new StringWriter(); <----- Se construye el flujo de lectura
                                                          y los dos de escritura.
   StringWriter flujoEscMay = new StringWriter();
   int leido=0;
   while ( (leido = flujoLecCad.read()) != -1){
      if(Character.isLowerCase((char)leido)){
         flujoEscMin.append((char)leido);
      }else if(Character.isUpperCase((char)leido)){
         flujoEscMay.append((char)leido); <----- Se añade cada tipo de letra a la
                                                  cadena correspondiente mediante
                                                  los flujos de escritura.
      }
   }
   flujoEscMin.flush();
   flujoEscMay.flush(); <----- Añade a la cadena cualquier
                               carácter almacenado en el buffer
                               del flujo.
   String[] arrCad = new String[2];
   arrCad[0] = flujoEscMin.toString();
   arrCad[1] = flujoEscMay.toString();
   return arrCad;
}
```

Comentario: Dado que en este caso no se utilizan flujos que añadan elementos de *buffering*, no sería necesario el vaciado de los mismos mediante `flush()`. Sin embargo, como en la mayoría de los casos sí se utilizarán tales elementos, se introduce aquí esta función para facilitar su familiaridad.

Ejercicio 7.7:

Añada los tres métodos anteriores a una clase de nombre Flujos1 y escriba un pequeño programa para comprobar su funcionamiento.

Planteamiento: Los métodos especificados se añaden como métodos de instancia a la clase Flujos1. Para la realización de la prueba no es necesario incluir ningún atributo adicional y bastará con escribir un constructor que no lleve a cabo ninguna acción. El programa de prueba se escribirá en el método main() de la clase PruebaFlujos1. En este método se instanciará un objeto de la clase Flujos1 y se invocarán sobre él los distintos métodos para comprobar el funcionamiento de los mismos.

Solución: La clase Flujos1 quedaría como sigue:

```
public class Flujos1 {

    public byte[][] rellenarMatrizIdentidad(int dimension){
        if(dimension <= 0){
            throw new IllegalArgumentException("Argumento no válido.");
        }
        byte[][] matriz = new byte[dimension][];
        for(int i = 0; i < matriz.length; i++){
            ByteArrayOutputStream fila = new ByteArrayOutputStream(dimension);
            for(int j = 0; j < dimension; j++){
                fila.write(i == j ? 1 : 0);
            }
            try{ fila.close(); } catch(IOException ioE){}
            matriz[i] = fila.toByteArray();
        }
        return matriz;
    }

    public void mostrarMatrizConFlujos(byte[][] matriz) {
        for(int i = 0; i < matriz.length; i++){
            ByteArrayInputStream fila = new ByteArrayInputStream(matriz[i]);
            byte elemento;
            while( (elemento = (byte)fila.read()) != -1){
                System.out.print(elemento + " ");
            }
            System.out.println();
        }

    }

    public String[] ClasificadorMinMay(String cad) throws IOException{
        StringReader flujoLecCad = new StringReader(cad);
        StringWriter flujoEscMin = new StringWriter();
        StringWriter flujoEscMay = new StringWriter();
        int leido=0;
        while ( (leido = flujoLecCad.read()) != -1){
            if(Character.isLowerCase((char)leido)){
                flujoEscMin.append((char)leido);
            }else if(Character.isUpperCase((char)leido)){
                flujoEscMay.append((char)leido);
            }
        }
        flujoEscMin.flush();
        flujoEscMay.flush();
```

```
            String[] arrCad= new String[2];
            arrCad[0] = flujoEscMin.toString();
            arrCad[1] = flujoEscMay.toString();
            return arrCad;

        }
    }
```

Por su parte, la clase PruebaFlujos1 es la siguiente:

```
public class PruebaFlujos1 {
    public static void main(String args[]){
        Flujos1 f1 = new Flujos1();

        byte[][] matByte1 = f1.rellenarMatrizIdentidad(4);
        f1.mostrarMatrizConFlujos(matByte1);

        try{
            String[] arrCad = f1.ClasificadorMinMay("aaaBBBaaJJFFEEjjjjlllA");
            System.out.println(Arrays.toString(arrCad));
        }catch(IOException ioE){
            System.out.println(ioE.toString());
        }
    }
}
```

Se obtiene una matriz identidad de dimensión 4 x 4.

Se muestra por pantalla la matriz identidad.

Con la clase Arrays se muestra el array obtenido que albergará una cadena en minúsculas y otra en mayúsculas.

Ejercicio 7.8:

Escriba un método, de nombre conversorMinMay, que reciba por parámetro un array de caracteres. El método devolverá un array de caracteres de forma que las letras minúsculas del array recibido figuren en el nuevo array como mayúsculas y el resto de caracteres permanezca sin cambios. El método no capturará ninguna excepción.

Planteamiento: El método crea un flujo para la lectura de los caracteres almacenados en el array recibido por parámetro que consistirá en un objeto de la clase CharArrayReader. Así mismo, el método habilita un flujo que permita la escritura de caracteres a un array creando un objeto de la clase CharArrayReader. Dentro de un bucle se procede a la lectura de los caracteres presentes en el array recibido. En cada iteración se comprueba si el carácter leído corresponde a una letra con el método isLetter() de la clase Character. En caso afirmativo la letra se convierte en mayúscula utilizando el método toUpperCase() de la misma clase. Finalmente la letra en mayúscula o el carácter no alfabético se escribe en el flujo de escritura para que quede almacenado en el array que se devolverá al terminar el método.

Parámetros: Array de caracteres para convertir sus letras minúsculas en letras mayúsculas.

Valor de retorno: El método de vuelve un array de caracteres donde todas las letras que contenga estarán en mayúsculas.

Solución:

```
public char[] conversorMinMay(char[] cad) throws IOException{
   CharArrayReader lectura    = new CharArrayReader(cad);
   CharArrayWriter escritura = new CharArrayWriter();
   int c;
   while ((c = lectura.read()) != -1){
      if (Character.isLetter(c)){
```

Creación de flujos de lectura y escritura para acceder a los caracteres del array recibido y generar el array resultado.

```
            c = Character.toUpperCase(c);
        }
        escritura.write(c);
    }
    lectura.close();
    escritura.close();
    return escritura.toCharArray();
}
```

Devolvemos los caracteres escritos en el flujo como array de caracteres.

Comentario: No se comprueba si el carácter leído corresponde específicamente a una letra minúscula y, sencillamente, se procede a su conversión a mayúscula. Si el carácter leído es una letra mayúscula la conversión no tendrá consecuencias.

LECTURA Y ESCRITURA DE UN ARCHIVO

Ejercicio 7.9:

Escriba un programa que escriba los 100 primeros números naturales en un archivo de nombre C:/misArchivos/numNaturales.txt.

Planteamiento: El programa crea un flujo de escritura mediante un objeto de la clase `FileOutputStream`. A continuación, se escriben todos los valores en el flujo mediante un bucle utilizando el método `write()` de la clase `FileOutputStream`. Finalmente, se cierra el flujo de escritura.

Solución:

```
public class Escribe100Numeros{
    public static void main(String args[]){
        try{
            FileOutputStream archivoNumeros =
                        new FileOutputStream("C:/misArchivos/numeros.txt");
            for(int i = 0; i < 100; i++){
                archivoNumeros.write(i);
            }
            archivoNumeros.close();
        }catch(IOException ioE){
            System.out.println("Error de escritura: " + ioE.toString());
        }
    }
}
```

Creación del flujo de escritura para almacenar los valores en el archivo.

Ejercicio 7.10:

Escriba un método, de nombre obtenerSumaNumerosArchivo, que reciba por parámetro el nombre de un archivo que almacenará una serie de cantidades enteras y positivas. El método leerá todos los valores del archivo, calculará su suma y la devolverá. No se capturará ninguna excepción.

Planteamiento: El método define un flujo para la lectura del archivo recibido creando un objeto de la clase `FileInputStream`. Se utiliza un bucle para leer, en cada iteración, un valor del flujo utilizando el método `read()`. Cada valor se sumará sobre una variable que finalmente se devuelve.

Parámetros: Nombre del archivo que contiene los valores que se sumarán.

Valor de retorno: Valor entero correspondiente a la suma de todas las cantidades almacenadas en el archivo.

Solución:

```
public int obtenerSumaNumerosArchivo(String nombreArchivo) throws IOException{
   FileInputStream archivoNum = new FileInputStream(nombreArchivo);
   int numero, suma = 0;
   while ( (numero = archivoNum.read()) != -1){
      suma += numero;
   }
   archivoNum.close();
   return suma;
}
```

Aviso: Se requiere que el archivo almacene valores positivos porque el valor –1 se utiliza para determinar el fin del archivo. En los ejercicios siguientes la utilización de `Filtros` permite solucionar este problema.

Ejercicio 7.11:

Escriba un método, de nombre matrizAArchivo, que reciba una matriz con valores de tipo byte por parámetro y almacene su contenido linealmente en un archivo cuyo nombre también se recibirá por parámetro. La lectura de los elementos de la matriz se realizará con flujos de lectura sobre los elementos de su segunda dimensión.

Planteamiento: A partir del nombre de archivo recibido por parámetro se crea el correspondiente flujo de escritura representado por un objeto de la clase `FileOutputStream`. Mediante un bucle se recorren los elementos según la primera dimensión de la matriz, definiendo para cada uno de ellos un flujo de lectura como objeto de la clase `ByteArrayInputStream`. Los valores leídos de la matriz se escriben en el archivo, que finalmente se cierra mediante el método `close()`.

Parámetros: Matriz con valores de tipo `byte` que se almacenarán en el archivo y el nombre del archivo donde se guardarán.

Valor de retorno: El método no devuelve ningún valor de retorno, por lo que se utiliza la cláusula `void`.

Solución:

```
public void matrizAArchivo(byte[][] matriz, String nombreArchivo) throws IOException{
    FileOutputStream archivo =
            new FileOutputStream(nombreArchivo);   <-- Creación del flujo de escritura sobre el archivo.
    for(int i = 0; i < matriz.length; i++){
        ByteArrayInputStream fila =
            new ByteArrayInputStream(matriz[i]);
        byte elemento;
        while( (elemento = (byte)fila.read()) != -1){
            archivo.write(elemento);   <-- Cada valor que se lee de la matriz se escribe en el archivo.
        }
    }
    archivo.close();
}
```

Ejercicio 7.12:

Escriba un método, de nombre obtenerMatrizDeArchivo, que reciba por parámetro el nombre de un archivo que almacena linealmente los valores de tipo byte de una matriz. Se recibirán también las dimensiones de la matriz y el método la construirá y rellenará con los valores del archivo para finalmente, devolverla.

Planteamiento: Se crea un flujo de lectura sobre el archivo cuyo nombre se especifica por parámetro. A continuación, se comprueba que el número de valores almacenados en el archivo coincide con los necesarios para rellenar la matriz según las dimensiones recibidas. Se construye la matriz a rellenar a partir de dichas dimensiones. Mediante un bucle se van leyendo los valores almacenados en el archivo para ir asignándolo a las correspondientes posiciones de la matriz. La asignación se va realizando a lo largo del índice según la segunda la dimensión. Cuando dicho índice alcanza su límite, se reinicia al primer valor (0) a la vez que se incrementa el índice de la primera dimensión. Finalmente, se cierra el archivo de lectura y se devuelve la matriz.

Parámetros: Cadena de caracteres con el nombre del archivo del que se va a realizar la lectura de valores y dos valores enteros correspondientes a las dimensiones de la matriz a rellenar.

Valor de retorno: El método devuelve una matriz de valores `byte`, por tanto el valor de retorno será del tipo `byte[][]`.

Solución:

```
public byte[][] obtenerMatrizDeArchivo(String nombreArchivo, int dim1, int dim2)
                                        throws IOException{
    FileInputStream archivo =
          new FileInputStream(nombreArchivo); <---- Creación del flujo de lectura del archivo de entrada.
    if(archivo.available() != dim1*dim2){
          throw new IllegalArgumentException("Argumentos no correctos");
    }
    byte[][] matriz = new byte[dim1][dim2];
    int elemento,i = 0, j = 0;
    while( (elemento = archivo.read()) != -1){
       matriz[i][j++] = (byte)elemento;   <---- Asignación de los valores del archivo a la matriz.
       if(j == dim2){
          j = 0;
          i++;
       }
    }
    archivo.close();
    return matriz;
}
```

Ejercicio 7.13:

Escriba un método, de nombre escribirCadenasEnArchivo, que reciba un array de cadenas de caracteres y vuelque su contenido a un archivo cuyo nombre también se recibirá por parámetro. Las cadenas quedarán separadas en el archivo por un asterisco. El método no capturará ninguna excepción que pueda producirse.

Planteamiento: El método crea un flujo de escritura sobre el archivo recibido por parámetro creando un objeto de la clase `FileWriter`. Esta clase proporciona un método `write()` que permite la escritura de caracteres en el flujo. De esta forma se utiliza un bucle para ir recorriendo las cadenas almacenadas en el array recibido. Los caracteres de cadena se irán escribiendo en el archivo con ayuda del método anterior. Al final de cada cadena se escribe un asterisco en el archivo. Por último, se cierra el flujo sobre el archivo. Se crea un flujo de lectura sobre el archivo cuyo nombre se especifica por parámetro. A continuación, se comprueba que el número de valores almacenados en el archivo coincide con los necesarios para rellenar la matriz según las dimensiones recibidas. La matriz a rellenar es construida a partir de dichas dimensiones. Mediante un bucle se van leyendo los valores almacenados en el archivo para ir asignándolo a las correspondientes posiciones de la matriz. La asignación se va realizando a lo largo del índice según la segunda dimensión. Cuando dicho índice alcanza su

límite, es reiniciado al primer valor (0) a la vez que se incrementa el índice según la primera dimensión. Seguidamente, el archivo de lectura es cerrado. Finalmente, la matriz es devuelta.

Parámetros: Nombre del archivo sobre el que se escribirán las cadenas contenidas en el array que también se recibe.

Valor de retorno: El método no devuelve ningún valor de retorno.

Solución:

```
public void escribirCadenasEnArchivo(String nombreArchivo, String[] cadenas)
                                                   throws IOException{
   FileWriter escritura= new FileWriter(nombreArchivo);
   for(String cad: cadenas){
      for(int i = 0; i < cad.length(); i++){
         escritura.write(cad.charAt(i)); <-- Los caracteres de todas las cadenas se escriben en el archivo.
      }
      escritura.write('*');
   }
   escritura.close();
}
```

Comentario: Podría mejorarse la implementación del método utilizando el método `toCharArray()` de la clase `String` que permite obtener directamente un array con los caracteres que forman la cadena. De esta forma se evitaría tener que acceder individualmente a cada carácter de la cadena.

Ejercicio 7.14:

Escribir un método, de nombre LeerCadenasDeArchivo, que reciba por parámetro el nombre de un archivo y devuelva un array con las cadenas que dicho archivo almacena. Se tendrá en cuenta que las distintas cadenas se separan en el archivo por un asterisco. El método no capturará ninguna excepción que pueda producirse.

Planteamiento: El método crea un flujo de lectura sobre el archivo recibido por parámetro mediante un objeto de la clase `FileReader`. Dado que el número de cadenas almacenado por el archivo no puede conocerse a priori, se utiliza un objeto `ArrayList` para ir almacenando las cadenas leídas. Tampoco es posible determinar inicialmente la longitud de cada una de las cadenas almacenadas, por lo que se utiliza un flujo de escritura `CharArrayWriter` para ir almacenando los caracteres correspondientes a cada cadena. De esta forma se van leyendo caracteres del archivo mediante el método `read()` que se van escribiendo en el flujo de escritura `CharArrayWriter`. Cuando se encuentra un asterisco, este flujo se cierra y mediante el método `toString()` se convierte en una cadena que puede ser almacenada en el objeto `ArrayList`. Además, el flujo de escritura vinculado al array de caracteres correspondientes a cada cadena es reseteado para que almacene los caracteres correspondientes a la siguiente cadena. Finalmente, se obtiene un array de cadenas a partir del objeto `ArrayList` que será devuelto.

Parámetros: Nombre del archivo del que se leerán las cadenas de caracteres.

Valor de retorno: Array con las cadenas leídas del archivo.

Solución:

```
public String[] leerCadenasDeArchivo(String nombreArchivo) throws IOException{
   FileReader archive = new FileReader(nombreArchivo);
   ArrayList<String> al = new ArrayList<String>();
   int lectura;
```

```
    CharArrayWriter cad = new CharArrayWriter();
    while( (lectura = archivo.read()) != -1 ){
        char car = (char)lectura;
        if(car == '*'){
            cad.close();
            al.add(cad.toString());
            cad.reset();
        }else{
            cad.write(car);
        }
    }
    return ((String[])al.toArray(new String[al.size()]));
}
```

Flujo de escritura para almacenar los caracteres de cada cadena.

Cuando se encuentra un asterisco el flujo de escritura se convierte a una cadena de caracteres que se añade al objeto `ArrayList`.

Comentario: Como se verá en la siguiente sección, la lectura desde un archivo resulta mucho más sencilla dotando al flujo de lectura de un elemento de buffer que permita la lectura de cadenas completas. Consulte el Ejercicio 7.15.

FILTROS

Ejercicio 7.15:

Escriba un método, de nombre obtenerFilasMatrizChar, que reciba por parámetro una matriz de caracteres. El método devolverá un array con las cadenas de caracteres correspondientes a los elementos situados a lo largo de la segunda dimensión (filas) de la matriz.

Planteamiento: Este método pretende ilustrar la facilidad de lectura de líneas completas (hasta encontrar una marca de fin de línea o de fin de archivo) que ofrece la clase `BufferedReader`. En este caso, el funcionamiento del método consiste en recorrer secuencialmente los elementos de la matriz según la primera dimensión. Por su condición de matriz, cada uno de estos elementos consistirá a su vez en un array de caracteres. Sobre cada uno de estos arrays se construye un flujo de lectura utilizando la clase `CharArrayReader` que se complementa por el buffer añadido por la clase `BufferedReader`. Utilizando el método `readLine()` de esta última clase se puede leer directamente una cadena con todos los caracteres correspondientes a una fila de la matriz. El algoritmo funciona correctamente porque el método `readLine()` obtiene en una única cadena todos los caracteres disponibles en el flujo sobre el que se ha construido, hasta alcanzar una marca de fin de línea o el fin del flujo. En este ejemplo, cada lectura devolverá la cadena correspondiente a la concatenación de todos los caracteres de una fila de la matriz. El array con todas las cadenas será finalmente devuelto.

Parámetros: Matriz de caracteres para ser leer directamente cada una de sus filas mediante un flujo de lectura dotado de buffer.

Valor de retorno: Array de cadenas, donde cada una de ellas contiene los caracteres correspondientes a una fila de la matriz.

Se dota al flujo de lectura de un buffer para permitir la lectura de varios caracteres simultáneamente.

Solución:

```
public String[] obtenerFilasMatrizChar(char[][] matrizCar)throws IOException{
    String[] arrCadenas=new String[matrizCar.length];
    for(int i = 0; i < matrizCar.length; i++){
        BufferedReader flectura= new BufferedReader(new CharArrayReader(matrizCar[i]));
        arrCadenas[i]=flectura.readLine();
    }
```

```
        return arrCadenas;
    }
```

Ejercicio 7.16:

Escriba un método, de nombre escribirMes, que reciba por parámetro el número correspondiente a un mes del año y la letra correspondiente al día de la semana en que cae el día 1 de ese mes ('L' para Lunes, 'M' para Martes, 'X' para Miércoles...). El método escribirá en un archivo un calendario para ese mes que estará formado por todos los días del mes seguidos de la letra correspondiente al día de la semana en que caen. El método comprobará que los parámetros recibidos son correctos y obtendrá el nombre concatenando a la palabra "mes" el número de mes recibido más la extensión ".txt".

Ejemplo: Supóngase que el método recibe el valor 3 (representa el mes Marzo) y la letra 'M' indicando que el 1 de Marzo es Martes. El método obtendrá el nombre de archivo "mes3.txt" y lo escribirá con un calendario que tendrá un aspecto como éste:

1M2X3J5V6S7D8L. . . 31J

Planteamiento: El método comprueba, en primer lugar, que los parámetros recibidos son correctos. A continuación, crea un flujo de escritura sobre el nombre de archivo obtenido. El flujo de escritura lo representa un objeto de la clase `DataOutputStream`, ya que esta clase actúa como filtro añadiendo métodos que permiten la escritura de tipos de datos por separado. El objeto `DataOutputStream` parte de un objeto `BufferedOutputStream`, que tiene la particularidad de añadir un sistema de almacenamiento intermedio (*buffering*) al flujo inicial establecido sobre un archivo. Después se determina el número de días del mes considerando un año no bisiesto y este número se utiliza para controlar un bucle que, en cada iteración, escribe en el archivo un número de día seguido de su correspondiente letra. El archivo es finalmente cerrado.

Parámetros: Un valor entero indicando el mes del año y una letra para indicar el día de la semana correspondiente al día uno de dicho mes.

Valor de retorno: El método no devuelve ningún valor de retorno.

Solución:

```
public void escribirMes(int mes, char letraDíaUno) throws IOException{
    if( mes<1 || mes>12){
        throw new IllegalArgumentException("Mes no válido.");
    }
    String letraDía1 = "" + letraDíaUno;  <-- Se pasa el carácter a cadena para transformarlo a mayúscula antes de buscar en la cadena con las letras válidas.
    String letrasDíasSemana = "LMXJVSD";
    if (letrasDíasSemana.indexOf(letraDía1.toUpperCase()) == -1){
        throw new IllegalArgumentException("Letra día 1 no válida.");
    }
    DataOutputStream archivoMes = new DataOutputStream(
                    new BufferedOutputStream(
                    new FileOutputStream("C:/misArchivos/mes" + mes + ".txt")));
    int numDíasMes = 0;
    switch(mes){
        case 2: numDíasMes=28;  <-- Obtención número de días del mes.
                break;
        case 4:
        case 6:
        case 9:
        case 11:numDíasMes=30;
```

```
            break;
         default:numDíasMes=31;
      }

      for(int i = 0; i < numDíasMes; i++){
         char letra=letrasDíasSemana.charAt((letrasDíasSemana.indexOf(letraDíaUno)+i)%7);
         archivoMes.writeInt(i+1);
         archivoMes.writeChar((char)letra);
      }
      archivoMes.close();
   }
```

Se escribe en el archivo mediante métodos específicos para los distintos tipos primitivos.

Comentario: Para calcular la letra correspondiente a cada día se comienza con la posición que ocupa la correspondiente al día 1 en la cadena `letrasDíasSemana`. A esta posición se le suma el día pero, a continuación, se aplica el módulo de la división entre siete para no desbordar la cadena. En el archivo se escribe `i+1` porque interesa comenzar desde 1 para calcular correctamente la letra correspondiente a cada día. Por supuesto, como no existe el día 0, se incrementa la variable para que la primera escritura sea la correspondiente al día 1.

Ejercicio 7.17:

Escriba un método, de nombre consultarDíaMes, que reciba por parámetro un número de mes y un número de día. El método buscará en el archivo correspondiente al mes el día de semana (Lunes, Martes, Miércoles, ...) en que cae el día especificado. Para obtener el nombre del archivo a buscar, el método realizará el mismo procedimiento que en el ejercicio anterior. Se devolverá una cadena de caracteres con una frase explicativa del día de semana concreto. Si se solicita consultar un día que no corresponde al mes indicado se indicará devolviendo la correspondiente cadena.

Planteamiento: No es necesario validar la corrección de los parámetros. Si el número de mes no es correcto no se encontrará el correspondiente archivo y si el día no corresponde al mes, la búsqueda dentro del archivo fallará. Se va leyendo del archivo cada número y cada carácter con los correspondientes métodos del objeto `DataInputStream` que implementa el flujo de lectura abierto para el archivo. Como en el ejercicio anterior este objeto utiliza un objeto `BufferedInputStream` para añadir un buffer al flujo inicial. Si durante la lectura se produce algún error o se alcanza el fin de archivo sin encontrar la cadena, la correspondiente excepción se capturará y el método devolverá un mensaje notificando que el día no corresponde al mes especificado. Cuando se encuentre en el archivo el día buscado se utiliza la letra que le sigue para determinar a qué día corresponde, salvo el miércoles, el resto de letras que coinciden con la primera del nombre del día. Finalmente se devuelve la correspondiente cadena indicando el día de la semana.

Parámetros: Dos valores enteros correspondientes, respectivamente, al mes y al día que se quieren consultar.

Valor de retorno: Cadena de caracteres informando del día de la semana correspondiente al día consultado.

Solución:

```
public String consultaDíaMes(int mes, int día) throws IOException{
   DataInputStream archivoMes=new DataInputStream(
                              new BufferedInputStream(
                              new FileInputStream("C:/misArchivos/mes"+mes+".txt")));
   String[] meses={"Ene", "Feb", "Mar", "Abr", "May", "Jun",
                   "Jul", "Ago", "Sep", "Oct", "Nov", "Dic"};
   String[] díasSemana={"Lun", "Mar", "Mie", "Jue", "Vie", "Sab", "Dom"};
```

```
    int díaLeído  = 0;
    char letraDía = 0;
    boolean encontrado = false;
    try{
       while( !encontrado){
          díaLeído = archivoMes.readInt();
          letraDía = archivoMes.readChar();
          System.out.println("Día leído " + díaLeído);
          System.out.println("Letra: " + letraDía);
          if(díaLeído == día){
             encontrado = true;
          }
       }
    }catch(IOException ioE){
       return "El día " + día + "no corresponde al mes " + meses[mes-1];
    }
    if(encontrado){
       if (letraDía == "X"){
          return "El día " + día + "del mes " + meses[mes-1] + "es " + díasSemana[2];
       }else{
          for(String cad: díasSemana){
             if(letraDía == cad.charAt(0)){
                return "El día " + día + "del mes " + meses[mes-1] + "es " + cad;
             }
          }
       }
    }

    return "El día " + día + "no corresponde al mes " + meses[mes-1];
}
```

Los datos del archivo se leen con los métodos apropiados.

Cadena que se devuelve. Se resta 1 porque el primer mes en el array ocupa la posición cero.

Se busca en el array de díasSemana el nombre del día cuya primera letra coincida con la correspondiente al día consultado.

Ejercicio 7.18:

Escriba un método, de nombre escribirCadenasEnArhivo, que reciba por parámetro un array de cadenas de caracteres y el nombre de un archivo. El método volcará el contenido del array de cadenas en el archivo especificado. El método no capturará ninguna excepción.

Planteamiento: El método comienza creando un flujo de escritura sobre el nombre de archivo especificado. Para obtener dicho flujo se crea un objeto `PrintStream` a partir de un flujo adecuado para la escritura de caracteres Unicode y representado por un objeto `FileWriter`. El método recorre las cadenas almacenadas en el array mediante un bucle de forma que en cada iteración se almacene una cadena en el archivo. Finalmente se cierra el flujo de escritura.

Parámetros: Array con cadenas de caracteres que serán almacenadas en el archivo cuyo nombre también se especifica.

Valor de retorno: El método no devuelve ningún valor de retorno.

Solución:

```
public void escribirCadenasEnArchivo(String[] cadenas, String nombreArchivo)
                          throws IOException{
    PrintWriter flujoEscritura = new PrintWriter(
```

```
                                      new BufferedWriter(
                                          new FileWriter(nombreArchivo)));
    for(String cad : cadenas){
        if (cad != null){
            flujoEscritura.println(cad); <-------- Escribe en el archivo todas las cadenas del array en una nueva línea.
        }
    }
    flujoEscritura.close();
}
```

Avisos: Se utiliza el método `println()` para dejar una marca de fin de línea cuando se escriba cada línea en el archivo. Esto facilitará su lectura posterior utilizando el método `readLine()`.

Ejercicio 7.19:

Escriba un método, de nombre obtenerLíneasArchivo, que reciba por parámetro el nombre de un archivo de texto y devuelva todas sus líneas en un array de cadenas de caracteres. El método no capturará ninguna Excepción.

Planteamiento: El método comienza creando un flujo de lectura sobre el nombre del archivo recibido. Para ello se utiliza la clase `FileReader` que permite la lectura de caracteres Unicode y, además, la clase `BufferedReader` que dotará de un buffer al flujo de lectura de forma que, entre otras cosas, puedan leerse líneas completas. Cada línea del archivo se lee en una iteración del bucle utilizado a tal efecto y se almacena en un objeto `ArrayList` que, finalmente, se utilizará para devolver un array con las cadenas que almacene.

Parámetros: Nombre del archivo cuyas líneas serán devueltas en un array de cadenas de caracteres.

Valor de retorno: Array de objetos de la clase `String` con las líneas del archivo recibido.

Solución:

```
public String[] obtenerLineasDeArchivo(String nombreArchivo) throws IOException{
    BufferedReader flujoLectura = new BufferedReader(new FileReader(nombreArchivo));
    String lectura = null;
    ArrayList<String> al = new ArrayList<String>();
    while( (lectura = flujoLectura.readLine()) != null ){ <-------- Lectura de las líneas del archivo.
        al.add(lectura);
    }
    flujoLectura.close();
    return ((String[])al.toArray(new String[al.size()]));
}
```

Aviso: El método `readLine()` lee todos los caracteres del archivo situados antes de una marca de fin de línea (carácter '\n') sin incluirla. Por tanto es necesario que las líneas del archivo se hayan escrito dejando esta marca al final de cada una de ellas.

Ejercicio 7.20:

Escriba un método, de nombre mostrarArhivoPantalla, que reciba por parámetro el nombre de un archivo de texto y muestre su contenido por pantalla. El método no capturará ninguna excepción.

Planteamiento: El método abre un flujo de lectura mediante un objeto de la clase `BufferedReader`. Mediante un bucle se lee cada una de las líneas del archivo y se muestra por pantalla.

Parámetros: Nombre del archivo de texto cuyas líneas se mostrarán por pantalla.

Valor de retorno: El método no devuelve ningún valor de retorno.

Solución:

```
public void mostrarArchivoPantalla(String nombreArchivo) throws IOException{
    BufferedReader flujoLectura = new BufferedReader(new FileReader(nombreArchivo));
    String lectura = null;
    while( (lectura = flujoLectura.readLine()) != null ){ ← Lectura de las líneas del archivo.
        System.out.println(lectura);
    }
    flujoLectura.close();
}
```

Ejercicio 7.21:

Escriba un método, de nombre rellenarMatrizDesdeArchivo, que reciba por parámetro el nombre de un archivo de texto y una matriz de caracteres. El método rellenará la matriz con los caracteres almacenados en el archivo y la devolverá. El método no capturará ninguna excepción.

Planteamiento: El método abre un flujo de lectura al archivo recibido mediante un objeto de la clase `BufferedReader`. Se utiliza el método `read()` para volcar los caracteres del archivo a los elementos en la segunda dimensión de la matriz, es decir, la matriz se rellena por filas. Este método permite especificar en su primer argumento el array de caracteres donde se volcarán los caracteres leídos del archivo. El segundo parámetro indica un desplazamiento respecto de la posición de lectura, en el ejercicio se deja a valor 0 porque se asume que los caracteres están almacenados linealmente en el archivo. El último argumento indica el número de caracteres que se intentará leer del archivo. La matriz rellena finalmente se devuelve.

Parámetros: Nombre del archivo de texto cuyas líneas se mostrarán por pantalla.

Valor de retorno: El método no devuelve ningún valor de retorno.

Solución:

```
public char[][] rellenarMatrizDesdeArchivo(char[][] matriz, String nombreArchivo)
                        throws IOException{
    BufferedReader archivo = new BufferedReader(new FileReader(nombreArchivo));
    String lectura;
    boolean puedeLeer = true;
    for(int i = 0; i < matriz.length && puedeLeer; i++){
        puedeLeer = (archivo.read(matriz[i],0,matriz[0].length) !=-1 ); ← read() lee directamente de las filas de la matriz un número de caracteres del archivo igual a su longitud.
    }
    lectura.close();
    return matriz;
}
```

Aviso: Si el archivo almacena menos caracteres de los necesarios para rellenar la matriz, el bucle de lectura de las filas de la matriz se interrumpirá y habrá posiciones cuyo contenido no se modifique. Si ocurre al contrario, se dejará de leer del archivo cuando se haya rellenado la última fila de la matriz. Debe tenerse en cuenta que caracteres como el de fin de línea serán volcados sobre la matriz.

Ejercicio 7.22:

Escriba un método, de nombre clasificadorPalabrasLongitud, que reciba por parámetro el nombre de un archivo que contendrá un conjunto de palabras, cada una en una línea. El método recibirá también un valor entero que se utilizará como valor de corte para clasificar las palabras del archivo anterior. Se recibirán

también los nombres de los archivos donde quedarán almacenadas, respectivamente, las palabras con longitud menor al valor de corte y el resto de palabras. El método no capturará ninguna excepción.

Planteamiento: Se crea un flujo de lectura sobre el archivo recibido al que se dota de un buffer. Este flujo quedará representado por un objeto de la clase `BufferedReader`. A continuación, se crean dos flujos de escritura sobre los archivos que contendrán las palabras clasificadas en función de su longitud. Para la creación de estos flujos se utilizan las clases `BufferedWriter` que permiten dotar al flujo de un elemento de *buffering* y la clase `PrintWriter` que facilitará la escritura en el flujo a través de su método `println()`. Se procede a la lectura de todas las palabras del archivo especificado para, según su longitud, escribirlas en uno u otro archivo de clasificación. Finalmente se cierran todos los flujos abiertos.

Parámetros: Una cadena de caracteres correspondiente al nombre de un archivo que almacena las palabras que van a clasificarse en función de su longitud. Un valor entero utilizado como valor de corte en la clasificación y dos cadenas de caracteres correspondientes a los nombres de los archivos donde quedarán almacenadas, respectivamente, las palabras del primer archivo con longitud inferior al valor de corte y el resto de palabras.

Valor de retorno: El método no devuelve ningún valor de retorno.

Solución:

Creación de los flujos de lectura y escritura.

```
public void clasificadorPalabrasLongitud(String nomArchivo,
                                          int longitudCorte,
                                          String nomArchivoPalMenorL,
                                          String nomArchivoPalMayorL) throws IOException{
    BufferedReader entrada = new BufferedReader (new FileReader(nomArchivo));
    PrintWriter salida1 = new PrintWriter(new BufferedWriter(
                                   new FileWriter(nomArchivoPalMenorL)));
    PrintWriter salida2 = new PrintWriter(new BufferedWriter(
                                   new FileWriter(nomArchivoPalMayorL)));
    String lectura;
    while( (lectura = entrada.readLine()) != null){
        if (lectura.length() < longitudCorte){
            salida1.println(lectura);
        }else{
            salida2.println(lectura);
        }
    }
   entrada.close();
   salida1.close();
   salida2.close();
}
```

Se escribe cada sobre el flujo de escritura adecuado en base a su longitud.

Ejercicio 7.23:

Escriba un método, de nombre ordenarArchivoAlfab, que reciba por parámetro el nombre de un archivo para dejar sus líneas ordenadas alfabéticamente. La ordenación se realizará sin tener en cuenta que las letras en mayúsculas preceden a las minúsculas en el código Unicode.

Planteamiento: El método comienza creando un flujo de lectura sobre el archivo recibido y un objeto de la clase `ArrayList<String>` para almacenar todas sus líneas. Mediante el correspondiente bucle se procede a la lectura de todas las líneas del archivo y a su almacenamiento en la lista creada. Una vez concluida la lectura se cierra el flujo y se ordena la lista con ayuda del método `sort()` de la clase `Collections`. Se modifica el criterio

de ordenación correspondiente al orden natural establecido para la clase `String` especificando que no se distinga entre letras mayúsculas y minúsculas. A continuación se abre un flujo de escritura sobre el mismo archivo recibido y se procede a escribir sobre él el contenido de la lista ordenada. Finalmente se cierra el flujo de escritura.

Parámetros: Cadena de caracteres correspondiente al nombre de un archivo cuyas líneas van a quedar ordenadas alfabéticamente.

Valor de retorno: El método no devuelve ningún valor de retorno.

Solución:

```
public void ordenarArchivoAlfab(String nombreArchivo) throws IOException{
    BufferedReader entrada = new BufferedReader (new FileReader(nombreArchivo));
    ArrayList<String> al= new ArrayList<String>();
    String lectura=null;
    while( (lectura=entrada.readLine())!=null){
        al.add(lectura);
    }
    entrada.close();
    Collections.sort(al, String.CASE_INSENSITIVE_ORDER);
    PrintWriter salida= new PrintWriter(new BufferedWriter(
                                    new FileWriter(nombreArchivo)));
    ListIterator<String> l = al.listIterator();
    while (l.hasNext()){
        salida.println(l.next());
    }
    salida.close();
}
```

ArrayList donde volcar las líneas del archivo para después ordenarlas.

Las líneas ordenadas se escriben en el mismo archivo. El archivo original se modifica.

Ejercicio 7.24:

Escriba un método, de nombre comprobarArchivoMatriculas, que reciba por parámetro el nombre de un archivo que contendrá una serie de matrículas de automóviles. El método eliminará del archivo aquellas matrículas que no cumplan el sistema de numeración vigente. Este sistema de numeración especifica que cada matricula estará compuesta por tres letras en mayúsculas que no pueden ser vocales seguidas de un espacio en blanco y cuatro dígitos.

Planteamiento: En primer lugar, se procede a la creación de un flujo que permita la lectura de las matrículas almacenadas en el archivo recibido. Se crea también una lista para almacenar las matrículas cuya numeración cumpla el formato especificado en el enunciado. Dentro de un bucle se va leyendo cada matrícula almacenada en el archivo. Se utiliza el método `match()` de la clase `Pattern` para verificar si cada matrícula leída presenta el formato adecuado, en caso afirmativo la matrícula se añadirá a la lista. Una vez procesadas todas las matrículas del archivo, se cierra el flujo de lectura y se abre otro de escritura sobre el mismo archivo para volcar las matrículas correctas. Finalmente se cierra también el flujo de escritura.

Parámetros: Nombre del archivo con las matrículas de automóviles a verificar.

Valor de retorno: El método no devuelve ningún valor de retorno.

Solución:

```
public void comprobarArchivoMatriculas(String nombreArchivo) throws IOException{
    BufferedReader entrada = new BufferedReader (new FileReader(nombreArchivo));
```

```
        ArrayList<String> al= new ArrayList<String>();
        String lectura=null;
        while( (lectura=entrada.readLine())!=null){
            if(Pattern.matches("[0-9]{4}\\s[A-Z&&[^AEIOU]]{3}", lectura))
                al.add(lectura);
        }
        entrada.close();
        PrintWriter salida= new PrintWriter(new BufferedWriter(
                                           new FileWriter(nombreArchivo)));
        ListIterator<String> l = al.listIterator();
        while (l.hasNext()){
            salida.println(l.next());
        }
        salida.close();
    }
```

Construcción de la expresión regular que valida las matriculas.

Las matriculas con formato correcto se guardan en el objeto ArrayList para luego volcarlas al archivo.

Comentario: La interpretación de la expresión regular es la siguiente: `[A-Z&&[^AEIOU]]` indica cualquier letra mayúscula a excepción de las vocales (que no pueden formar parte de una matrícula), `{3}` indica que deben figurar 3 de las letras anteriores en la expresión. La secuencia `\\s` indica un carácter en blanco y `[0-9]{4}` indica exactamente cuatro dígitos del 0 al 9.

Entrada desde teclado

Ejercicio 7.25:

Escriba un método, de nombre pedirEntero, que solicite al usuario un valor entero. El método no dejará de solicitarlo hasta que el usuario introduzca un valor entero que se encuentre, además, dentro del rango correspondiente a este tipo de datos.

Planteamiento: El método crea un flujo de lectura a partir del objeto estático `in` de la clase `System`. Se utiliza la clase `InputStreamReader` para que el flujo permita la lectura directa de caracteres Unicode. Se utiliza además la clase `BufferedReader` para dotar al flujo de un buffer y permitir la lectura de líneas completa. El método recibe un mensaje que es mostrado por pantalla para invitar al usuario a que introduzca un valor entero. El valor es leído con ayuda del método `readLine()` y se trata de convertir al tipo de datos entero utilizando el método `parseInt()` suministrado por la clase `Integer`. Las excepciones que puedan originarse en este proceso de lectura y conversión son recogidas para informar al usuario de la ocurrencia del error y hacer que se le vuelva a solicitar de nuevo el valor.

Parámetros: Mensaje para solicitar un número entero al usuario. (*Ejemplo: Introduzca su edad.*)

Valor de retorno: Valor entero introducido por el usuario.

Solución:

```
public int pedirEntero(String mensaje){
    BufferedReader teclado = new BufferedReader(new InputStreamReader(System.in));
    int entero=0;
    boolean correcto=true;
    do{
        System.out.println(mensaje);
        try{
            entero = Integer.parseInt(teclado.readLine());
            correcto = true;
```

Lectura de una cadena desde teclado e intento de conversión a tipo de datos entero.

```
        }catch(IOException ioE){
          System.out.println("Error de entrada/salida");
          System.out.println("Error: "+ ioE.toString());
          correcto = false;
        }catch(NumberFormatException numForE){
          System.out.println("Error de conversion.Debe introducir un num. entero");
          System.out.println("Error: "+ numForE.toString());
          correcto = false;
        }
    }while(!correcto);
    return(entero);
}
```

Si se genera una excepción no se pudo leer correctamente el entero con lo que se solicitará de nuevo.

Ejercicio 7.26:

Escriba un método, de nombre pedirConfirmación, que solicite al usuario una respuesta a una pregunta cuyo contenido se recibirá por parámetro. El método continuará formulando la pregunta hasta que la respuesta consista en uno de los siguientes caracteres: 's', 'S', 'n', 'N'. En caso de responder afirmativamente ('s' o 'S') el método devolverá el valor true, y false en caso contrario.

Planteamiento: Como en el ejercicio anterior, el método crea un flujo de lectura dotado de buffer para leer caracteres del teclado. El método formula la pregunta hasta que se introduzca algunos de los caracteres que constituyen una respuesta válida. En ese caso, se devolverá `true` o `false` en función de la respuesta dada por el usuario.

Parámetros: Mensaje con la pregunta a formular al usuario.

Valor de retorno: Valor lógico *true* si el usuario ha respondido afirmativamente y *false* en caso contrario.

Solución:

```
public static boolean pedirConfirmacion(String mensaje){
    BufferedReader teclado = new BufferedReader(new InputStreamReader(System.in),1);
    String respuesta;
    boolean correcto = true;
    do{
        System.out.println(mensaje);
        try{
            respuesta=teclado.readLine().toUpperCase();
            if (!respuesta.equals("S") && !respuesta.equals("N"))
                throw new Exception("Debe introducir S(s) o N(n)");
            correcto = true;
        }catch(IOException ioE){
            System.out.println("Error de entrada/salida");
            System.out.println("Error: "+ ioE.toString());
            correcto = false;
        }catch(Exception confE){
            System.out.println("Error en su respuesta");
            System.out.println("Error: "+ confE.toString());
            correcto = false;
        }
    }while(!correcto);
    return(respuesta.equals("S")?true:false);
}
```

Conversión a mayúsculas para evitar la comparación con los caracteres en minúscula.

Ejercicio 7.27:

Escriba un método que presente al usuario un menú de 5 opciones. Se solicitará al usuario la elección de alguna de ellas hasta que su número de elección coincida con alguna de las opciones presentadas. El método devolverá el número de opción elegido por el usuario.

Planteamiento: El método presentará al usuario las opciones disponibles hasta que se elija una adecuada. La opción elegida será devuelta. Para solicitar el valor entero correspondiente a la opción deseada utilice el método `pedirEntero()` de la clase `EntradaTeclado` de la que se instancia un objeto.

Parámetros: El método no recibe ningún parámetro.

Valor de retorno: Valor entero correspondiente a la elección elegida por el usuario.

Solución:

```
public int menu(){
   EntradaTeclado et = new EntradaTeclado();
   int opcion;
   do{
       System.out.println("1. Opción 1");
       System.out.println("2. Opción 2");
       System.out.println("3. Opción 3");
       System.out.println("4. Opción 4");
       System.out.println("5. Salir");
       System.out.println();
       opcion=et.pedirEntero("Elija opción....");
   }while(opcion != 1 && opcion != 2 && opcion != 3 && opcion != 4 && opcion != 5);
   return opcion;
}
```

Comentario: La condición del `while` se podría haber escrito como:

```
}while (opcion < 1 || opcion  >  5);
```

Ejercicio 7.28:

Escriba un programa que pida al usuario cadenas de caracteres hasta que introduzca la cadena "fin" escrita en mayúsculas o minúsculas. Las cadenas introducidas por el usuario se almacenarán en un archivo de log con ruta "C:/misArchivos/log.txt". Cuando el usuario introduzca la cadena "fin" se mostrarán por pantalla todas las líneas introducidas previamente.

Planteamiento: Para poder realizar la lectura desde el teclado se utiliza un objeto de la clase `Scanner` pasándole como argumento en su creación el objeto estático `System.in` para leer del teclado. A continuación se abre un flujo de escritura dotado de buffer y con un filtro de la clase `PrintStream` sobre el archivo C:/misArchivos/log.txt. Dentro de un bucle se procede a leer líneas completas procedentes del teclado que, conforme se leen, se escriben con el flujo de escritura en el archivo. Se abandonará la ejecución del bucle cuando el usuario introduzca la cadena "fin" sin distinguir mayúsculas y minúsculas. En ese instante se cierra el flujo de escritura sobre el archivo de log, para abrir otro de lectura sobre el mismo archivo. El nuevo flujo estará controlado por la clase `Scanner` y con los métodos `hasNextLine()`, que devolverá cierto si quedan en el flujo más líneas por leer, y `nextLine()`, que devuelve la siguiente línea del flujo, se leen sucesivamente todas las líneas del archivo para mostrarlas por pantalla.

Solución:

```
public static void main(String[] args){
```

```
        Scanner lectorTeclado=new Scanner(System.in);
        PrintWriter log=null;

        try{
            log=new PrintWriter(new BufferedWriter(new FileWriter("C:/misArchivos/log.txt")));
            String lectura=null;
            while( !(lectura=lectorTeclado.nextLine()).toLowerCase().equals("fin")){
                System.out.println("Has dicho: " + lectura);
                log.println(lectura);
            }
            log.close();
        }catch(IOException ioE){
            System.out.println(ioE.toString());
        }

        try{
            System.out.println("Todo lo que dijiste.......");
            Scanner scan = new Scanner(new File("C:/misArchivos/log.txt"));
            while( scan.hasNextLine()){
                lectura = scan.nextLine();
                System.out.println("log: " + lectura);
            }
                scan.close();
            }catch(IOException ioE){
                System.out.println(ioE.toString());
            }
        }
    }
```

Flujo de lectura sobre el teclado.

Flujo de escritura en el archivo de log.

Flujo de lectura del archivo de log.

Ejercicio 7.29:

Escriba un programa que solicite al usuario números enteros hasta que:

- *La suma de todos los números introducidos sea superior a 100.*
- *Se introduzca 3 veces el valor 3.*
- *Se introduzca 2 veces el valor 2.*
- *Se introduzca 5 veces consecutivas el mismo valor.*

Planteamiento: Se construye un flujo de lectura sobre el teclado con la clase Scanner. Mediante un bucle se van solicitando valores enteros al usuario hasta que se cumpla alguna de las condiciones de terminación.

Solución:

```
public static void main(String[] args){
        Scanner lectorTeclado = new Scanner(System.in);
        String lectura = null;
        int repeticiones = 0;
        boolean primeraLectura = true;
        int numDoses  = 0;
        int numTreses = 0;
        int suma   = 0;
        int entero = 0, enteroAnterior = 0;
        do{
```

```
            System.out.println("Introduce un entero....");
            entero = lectorTeclado.nextInt();
            suma += entero;
            if(entero == 2){
                numDoses++;
            }
            if(entero == 3){
                numTreses++;
            }
            if(primeraLectura){
                System.out.println("paso por aqui");
                enteroAnterior = entero;
                primeraLectura = false;
            }
            if(entero == enteroAnterior){
                repeticiones++;
            }else{
                repeticiones = 1;
            }
            enteroAnterior = entero;
        }while( !(numDoses == 2 || numTreses == 3 || repeticiones == 5 ||suma > 100));
        System.out.println("Terminamos.....");

    }
}
```

La clase File

Ejercicio 7.30:

Para los siguientes ejercicios se aconseja crear primero un directorio o carpeta en la unidad C: del disco duro de nombre misArchivos. El directorio estaría ubicado directamente en la raíz de la unidad. Se pide escribir un programa que muestre los archivos y directorios situados en el directorio "C:/misArchivos.

Planteamiento: Se construye un objeto `File` con la ruta del directorio y se utiliza el método `list()` para obtener un array de `String` con los nombres de todos los archivos y directorios contenidos en él.

Solución:

```
public static void main(String args[]){
    File misArchivos  = new File("C:/misArchivos");
    String[] archivos = misArchivos.list();
    for(String cad: archivos){
        System.out.println(cad);
    }
}
```

Objeto `File` con la ruta del directorio a explorar.

Ejercicio 7.31:

Escriba un programa que muestre los nombres de los archivos con extensión ".java" contenidos en el directorio "C:/misArchivos"

Planteamiento: Para filtrar los nombres de archivos proporcionados por `list()` es necesario suministrarle un objeto de una clase que implemente la interfaz `FilenameFilter`. Esta interfaz define el método `accept()` que

devuelve un valor booleano para indicar si se incluye o no en la lista el nombre del archivo. El método `list()` incluye en el array que devuelve sólo los nombres para los que `accept()` devuelve `true`. Se implementa `accept()` de forma que sólo devuelva `true` si el nombre corresponde a un archivo (no a un directorio), tiene una longitud suficiente para terminar por la extensión ".java" y, por supuesto, termina efectivamente en esta extensión. Un objeto `File` representa un elemento del sistema de archivos. Por esta razón proporciona métodos como `getAbsolutePath()` que devuelve una cadena con la ruta absoluta, `isFile()` que indica si la ruta corresponde a un archivo o `isDirectory()` para indicar si se trata de un directorio.

Solución: Implementación de la clase que define el filtro sobre los nombres de archivo.

```
class FiltroArchivosJava implements FilenameFilter{
    public boolean accept(File dir, String name){
        boolean r = true;
        if ( new File(dir.getAbsolutePath()+"/" + name).isFile() &&
             name.length() >= 5 &&
             name.toLowerCase().lastIndexOf(".java") == name.length()-5){
            return true;
        }
        return false;
    }
}
```

Ruta del directorio más el nombre del archivo. Se comprueba que se trata de un archivo.

Se comprueba que el nombre de archivo termina con la cadena ".java".

Implementación del programa solicitado.

```
    public static void main(String args[]){
        File misArchivosJava  = new File("C:/misArchivos");
        String[] archivosJava = misArchivosJava.list(new FiltroArchivosJava());
        for(String cad: archivosJava){
            System.out.println("Java: " + cad);
        }
    }
```

Filtro para incluir en la lista sólo aquellos nombres de archivos que lo verifiquen.

Ejercicio 7.32:

Escriba un método, de nombre copiarArchivo, que reciba por parámetro dos rutas correspondientes a un archivo de origen y a un archivo de destino. El método copiará el archivo origen en la ubicación de destino. La ruta de destino puede consistir en un directorio o un archivo. En el primer caso, el archivo se copiará al directorio especificado manteniendo su nombre. En el segundo, se tomará como nombre del archivo copia el que especifique su ruta. Además el método recibirá un tercer parámetro que actuará de bandera en caso de que la ruta destino especifique un archivo y éste exista. Si la bandera es cierta el archivo destino será reemplazado por el que se copie. En caso contrario, la operación de copia terminará. El método generará una excepción DestinoProtegidoExcepcion si la bandera de reemplazo vale false y el archivo de destino ya existe.

Planteamiento: El método comprueba, en primer lugar, que las cadenas correspondientes a las rutas origen y copia no son nulas. A continuación se crean los objetos de tipo `File` a partir de dichas rutas. Se comprueba si la ruta de copia corresponde a un archivo y si éste existe. Así mismo, en el caso de que la ruta de copia especifique un directorio, se comprueba si existe en él un archivo con igual nombre que el archivo origen. En ambos casos se actúa en función de la bandera de reemplazo. Si el archivo destino de la copia existe y la bandera está a valor `false` la ejecución del método se abandona mediante el lanzamiento de la excepción `DestinoProtegidoExcepcion`. En otro caso se procederá a realizar la copia como si el archivo no existiera. El caso más sencillo es que la ruta de copia especifique un archivo destino inexistente o un directorio donde no se encuentre un archivo de igual nombre que el origen. En ambos casos, se procede a copiar el archivo origen en la ruta destino. Para llevar a cabo la copia, se leen, uno a uno, todos

los bytes del archivo origen que son escritos en el archivo copia. Finalmente se cierran los flujos sobre ambos archivos

Parámetros: Cadenas de caracteres correspondientes a las rutas de los archivos origen y copia. Bandera para indicar si el archivo destino será reemplazado en caso de que exista.

Valor de retorno: El método no devuelve ningún valor de retorno.

Solución:

```
class DestinoProtegidoExcepcion extends Exception{
    public DestinoProtegidoExcepcion(String msj){
        super(msj);
    }
}
public void copiarArchivo(String rutaOrigen, String rutaCopia, boolean reemplazar)
                                throws DestinoProtegidoExcepcion,
                                       FileNotFoundException,
                                       IOException{
   if(rutaOrigen == null ||
      rutaCopia  == null){
   throw new IllegalArgumentException("Argumentos no validos");
   }
   File origin = new File(rutaOrigen);
   File copia  = new File(rutaCopia);
   FileInputStream ent;
   FileOutputStream sal;
   if (copia.isFile() && !reemplazar ){
      throw new DestinoProtegidoExcepcion(
         "Error al copiar: El archivo destino ya existe y no se desea reemplazar");
   }
   if (copia.isDirectory()){
      copia=new File(copia.getAbsolutePath()+"/"+origen.getName());
      if (copia.isFile() && !reemplazar ){
         throw new DestinoProtegidoExcepcion(
            "Error al copiar: El archivo destino ya existe y no se desea reemplazar");
      }
   }
   try{
      ent = abrirArchivoLectura(rutaOrigen);
      sal = abrirArchivoEscritura(copia.getAbsolutePath());
   }catch(FileNotFoundException fnfE){
      throw new FileNotFoundException (
         "Error al copiar: No se pudo abrir algún archivo");
   }
   try{
      int leido;
      while((leido = ent.read()) != -1){
         sal.write(leido);
      }
      sal.close();
      ent.close();
```

Si la ruta de copia es un directorio, el archivo copia tendrá el mismo nombre que el original.

Se comprueba que el archivo copia exista tanto si la ruta de copia es un archivo como un directorio.

Los datos leídos del archivo de origen se escriben en el de copia.

```
        }catch(IOException ioE){
            throw new IOException("Error al copiar: Error de lectura/escritura");
    }
}
```

Comentario: La operación de copia sería más eficiente leyendo y escribiendo bytes en bloques para reducir el número de accesos al disco.

Ejercicio 7.33:

Escriba un método, de nombre moverArchivo, que reciba por parámetros dos rutas correspondientes a un archivo de origen y a un archivo de destino. El método moverá el archivo origen a la ubicación de destino. Como en el ejercicio anterior, la ruta destino puede consistir en un directorio o un archivo. En el primer caso, el archivo se moverá al directorio especificado manteniendo su nombre. En el segundo, además de cambiar de ubicación, el archivo será renombrado. Además el método recibirá un tercer parámetro que actuará de bandera en caso de que la ruta destino especifique un archivo y éste exista. Si la bandera es cierta el archivo destino será reemplazado por el archivo que se mueva. En caso contrario, la operación de cambio de ubicación terminará.

Planteamiento: Se comprueba que las cadenas recibidas no son nulas. Se obtienen las rutas del archivo fuente y destino y con ellas se invoca al método `copiarArchivo()`. Las excepciones que pueda lanzar el método `copiarArchivo()` son capturadas y relanzadas para indicar que el error se ha producido al invocarlo desde este método. Lo contrario resultaría desconcertante ya que invocando al método `moverArchivo()` se obtendrían excepciones correspondientes al proceso de copia. Si no se captura ninguna excepción, la operación de copia ha tenido éxito y procede a eliminar el archivo origen. Todas las comprobaciones relativas al carácter de archivo o directorio de la ruta destino son realizadas en el proceso de copia pues resultan similares.

Parámetros: Cadenas de caracteres correspondientes a las rutas origen y destino del cambio de ubicación de archivos. Bandera para indicar si el archivo destino será reemplazado en caso de que exista.

Valor de retorno: El método no devuelve ningún valor de retorno.

Solución:

```
public void moverArchivo (String rutaOrigen,
                          String rutaDestino,
                          boolean reemplazar) throws DestinoProtegidoExcepcion,
                                                     FileNotFoundException,
                                                     IOException{

    if(rutaOrigen  == null ||
       rutaDestino == null){
        throw new IllegalArgumentException("Argumentos no validos");
    }
    File origen  = new File(rutaOrigen);
    File destino = new File(rutaDestino);
    try{
        copiarArchivo(origen.getAbsolutePath(),   ← Se llama al método de copia de
                      destino.getAbsolutePath(),     archivos con las rutas absolutas.
                      reemplazar);
        origen.delete(); ←  Si la copia tiene éxito el archivo
                            origen se elimina.
    }catch(FileNotFoundException fnfE){
        throw new FileNotFoundException("Error al mover: No se pudo abrir algún archivo");
```

```
    }catch(IOException ioeE){
        throw new IOException("Error al mover: Error de lectura/escritura");
    }catch(DestinoProtegidoExcepcion dpE){
        throw new DestinoProtegidoExcepcion(
            "Error al mover: El archivo destino ya existe y no se desea reemplazar");
    }
}
```

Ejercicio 7.34:

Escriba un método que reciba las rutas correspondientes a dos directorios. El método copiará recursivamente el contenido del directorio origen en el directorio destino. La copia recursiva significa que si se encuentran subdirectorios en el directorio origen se copiarán también sus archivos y posibles subdirectorios al directorio destino.

Planteamiento: En primer lugar se comprueba que el directorio fuente exista, en caso contrario se lanza una excepción para su notificación. Se comprueba, también, si la ruta especificada como directorio destino existe, en cuyo caso el método termina elevando la correspondiente excepción para notificarlo. A continuación se procede a crear el directorio destino. Se obtiene la lista de archivos y directorios presentes en el directorio origen con ayuda del método `list()`. Los archivos son directamente copiados y los directorios son tratados como la copia del directorio actual, por lo que se realiza una llamada recursiva al método. Es importante tener en cuenta que la llamada recursiva debe realizarse especificando la ruta absoluta del subdirectorio origen y el subdirectorio destino. En caso de que se reciba alguna expcepción como resultado de la operación de copia, ésta se delegará abandonando así la ejecución del método.

Parámetros: Dos cadenas de caracteres con las rutas del directorio origen y el directorio destino.

Valor de retorno: El método no devuelve ningún valor de retorno.

Solución:

```
public void copiarDirectorio(String rutaOrigen, String rutaDestino)
                                                throws FileNotFoundException,
                                                       DestinoProtegidoExcepcion,
                                                       IOException,
                                                       Exception{

    if(rutaOrigen  == null ||
        rutaDestino == null){
            throw new IllegalArgumentException("Argumentos no validos");
    }
    File origen  = new File(rutaOrigen);
    File destino = new File(rutaDestino);
    if (!origen.isDirectory()){
        throw new Exception("No se pudo realizar la copia. El directorio origen no existe");
    }
    if (destino.isDirectory()){
        throw new Exception("No se pudo realizar la copia. El directorio destino ya existe");
    }

    String[] listaOrigen = origen.list();
    destino.mkdirs();
    for(String cad: listaOrigen){
        File f = new File(origen.getAbsolutePath() + "/" + cad);
```

```
            if(f.isFile()){
               copiarArchivo(f.getAbsolutePath(), destino.getAbsolutePath(), true);
            }
            else{
            File d = new File(destino.getAbsolutePath() + "/" + f.getName());
            copiarDirectorio(f.getAbsolutePath(), d.getAbsolutePath()); ←
            }
        }
    }
```

Llamada recursiva para la copia de subdirectorios.

Ejercicio 7.35:

Escriba un método, de nombre clasificarDirectorio, que reciba por parámetro una ruta de un directorio y una cadena correspondiente a la extensión de los archivos a clasificar. El método creará un subdirectorio con el mismo nombre de la extensión recibida y moverá a él todos los archivos con esta extensión.

Planteamiento: Se comprueba que las referencias recibidas no son nulas. Se verifica también que la referencia recibida como ruta de directorio corresponde efectivamente a un directorio. Se procede a determinar la lista de nombres de archivos que terminan con la extensión recibida. Para ello se suministra al método `list()` un objeto de una clase que implementa la interfaz `FilenameFilter`. En este caso se suministra una clase interna anónima. El método `accept()` valida los nombres si se corresponden con archivos, si su longitud es lo suficientemente grande para dar cabida a la extensión y, por último, si sus últimos caracteres coinciden con la extensión. Una vez obtenido el array de nombres, se comprueba que tiene algún elemento. En tal caso, se crea el directorio correspondiente a la extensión y todos los archivos de la lista son movidos del directorio especificado el directorio correspondiente a la extensión con ayuda del método `moverArchivo()`. Si la operación de movimiento de algún archivo genera alguna excepción, ésta es delegada abandonándose la ejecución del método.

Parámetros: Ruta de un directorio y cadena de caracteres correspondiente a la extensión de los archivos a clasificar.

Valor de retorno: El método no devuelve ningún valor de retorno.

Solución:

```
public void clasificarArchivos(String directorio, final String ext)
                                            throws FileNotFoundException,
                                            IOException,
                                            DestinoProtegidoExcepcion,
                                            Exception{
   if (directorio == null || ext == null ){
      throw new IllegalArgumentException("Argumento incorrecto");
   }
   final File dir = new File(directorio);
   if(!dir.isDirectory()){
      throw new Exception ("No se realizó la separación. " +
                           "Debe suministrarse un directorio");
   }

   String[] arExt = dir.list(new FilenameFilter(){ ←
                     public boolean accept(File d, String n){
                     return  new File(d.getAbsolutePath()+"/"+n).isFile() &&
                     n.length() > ext.length()+1 &&
                     n.toLowerCase().lastIndexOf("."+ext) ==
                     n.length()-ext.length()-1;
```

Filtro de nombres para los archivos con la primera extensión.

```
                        }
                    });
    if(arExt.length != 0){
        File dirExt = new File(dir.getAbsolutePath() + "/" + ext);
        dirExt.mkdirs();
        for(String nombre : arExt){
            File f = new File(dir.getAbsolutePath() + "/" + nombre);
            moverArchivo(f.getAbsolutePath(), dirExt.getAbsolutePath(), true);
    }
}
```

Creación del directorio correspondiente a la extensión.

Todos los archivos de la lista se mueven al directorio correspondiente a su extensión.

Archivos de acceso aleatorio

Ejercicio 7.36:

Escriba un programa que escriba 4 veces el valor 1 en un archivo y después 5 veces el valor 2. A continuación se mostrará el contenido del archivo por pantalla. Se continuará haciendo que todos los valores 2 del archivo se sustituyan por el valor 3. Se mostrará el nuevo contenido del archivo y el programa terminará.

Planteamiento: Se utiliza para este ejercicio un flujo de lectura/escritura abierto sobre un objeto de la clase RandomAccessFile. Una vez abierto el flujo se escribe 5 veces el valor 1 y 5 veces el valor 2. Los valores se escriben con el método writeInt(). A continuación, se pone a 0 el puntero de lectura/escritura del archivo con el método seek(). Se procede a recorrer el archivo para su lectura utilizando un bucle y el método readInt(). Se vuelve a resetear el puntero del archivo y, para sustituir todas las ocurrencias del valor 2 por el valor 3, se referencia la ocurrencia del primer valor 2 con el método seek(). Para indicar dicha posición se indica 5 enteros multiplicado por el tamaño en bytes del entero (4). Se escribe 5 veces el valor 3 y se vuelve a reiniciar el puntero para mostrar el contenido final del archivo. Por último, se cierra el archivo.

Solución:

Se abre el flujo para lectura y escritura.

```
public static void main(String[] args) {
    try{
        RandomAccessFile rf = new RandomAccessFile("C:/misArchivos/rf.txt", "rw");
        for(int i = 0; i < 10; i++){
            if(i < 5){
                rf.writeInt(1);
                }
                else{
                rf.writeInt(2);
                }
            }

        rf.seek(0);
        for(int i = 0; i < 10; i++){
            System.out.println("Valor: " + rf.readInt());
        }
        rf.seek(5 * 4);
        for(int i = 0; i < 10; i++){
            rf.writeInt(3);
        }

        rf.seek(0);
        for(int i = 0; i < 10; i++){
```

Se vuelve al inicio del archivo para imprimir sus valores.

Se posiciona el puntero tras 5 primeros enteros, es decir, en la posición del primer 2.

```
            System.out.println("Valor: " + rf.readInt());
        }
        rf.close();
    }catch(IOException ioE){
        System.out.println("Error: " + ioE.toString());
    }
}
```

Ejercicio 7.37:

Escriba un método, de nombre escribirArchivoPrecios, que reciba por parámetro un array de valores enteros correspondientes a referencias a artículos y un array de valores reales correspondientes a los precios de los artículos anteriores. El método recibirá también el nombre de un archivo sobre el que se escribirá cada referencia de artículo seguida de su precio. El método no capturará ninguna excepción.

Planteamiento: En primer lugar se crea un flujo para escribir las referencias y los precios en un archivo de acceso aleatorio. Mediante un bucle se escribe en el archivo cada referencia y su correspondiente precio. Finalmente se cierra el flujo de escritura sobre el archivo.

Parámetros: Array de valores enteros correspondientes a las referencias de los artículos y arrays de valores reales correspondientes a sus precios. También se recibe el nombre del archivo donde se volcarán alternativamente tanto las referencias como los precios.

Valor de retorno: El método no devuelve ningún valor de retorno.

Solución:

```
public void obtenerArchivoPrecios(int[] referencias,
                                  double[] precios,
                                  String nombreArchivo)
                    throws IOException{
    RandomAccessFile archive = new RandomAccessFile(nombreArchivo, "rw");
    for(int i = 0; i < referencias.length && i < precios.length; i++){
        archivo.writeInt(referencias[i]);
        archivo.writeDouble(precios[i]);
    }
    archivo.close();
}
```

Con los métodos `writeInt()` y `writeDouble()` se escriben los valores almacenados en los arrays al archivo.

Ejercicio 7.38:

Escriba un método, de nombre actualizarArchivoPrecios, que reciba por parámetro el nombre de un archivo que almacena un conjunto de parejas de valores correspondientes a una referencia de artículo y a su precio. El método actualizará los precios de forma que los superiores a 100 euros se decrementen en un 50% y los inferiores se incrementen en un 50%. El método capturará y tratará todas las excepciones que puedan producirse.

Planteamiento: En primer lugar se crea un flujo de lectura y escritura sobre el nombre del archivo recibido por parámetro. Alternativamente se van leyendo del archivo los precios y referencias almacenados en él y se procede a calcular la actualización del precio. Para reemplazar el precio leído por el actualizado se utiliza el método `seek()` para situarnos en la posición anterior al último precio leído. Para determinar dicha posición se utiliza el método `getFilePointer()` que proporciona la posición actual del puntero de lectura /escritura sobre el archivo y a este valor se le restan 8 posiciones correspondientes al número de bytes que ocupa cada valor de tipo `double`. Es importante tener en cuenta que todos los desplazamientos dentro del archivo se realizan en términos de bytes. A continuación, se escribe en el archivo el precio actualizado. El método utiliza la excepción `EOFException` para

determinar cuándo se han leído todos los valores del archivo. El resto de excepciones que se producen son capturadas y su tratamiento se limita a mostrar el correspondiente mensaje por pantalla.

Parámetros: Nombre del archivo cuyos precios van a ser actualizados.

Valor de retorno: El método no devuelve ningún valor de retorno.

Solución:
```
public void actualizarArchivoPrecios(String nombreArchivo){
        int referencia;
        double precio;
        boolean finArchivo = false;
        RandomAccessFile archive = null;
        try {
        archivo = new RandomAccessFile(nombreArchivo, "rw");
            do {
                try {
                    referencia = archivo.readInt();
                    precio = archivo.readDouble();
                    if (precio >= 100.0){
                        precio = precio*1.5;
                    }else{
                        precio = precio*0.5;
                    }
                    archivo.seek(archivo.getFilePointer()-8);
                    archivo.writeDouble(precio);
                } catch (EOFException e) {
                    finArchivo = true;
                    archivo.close();
                }
            } while (!finArchivo);
    } catch (FileNotFoundException e) {
        System.out.println("No se encontró el archivo.");
    } catch (IOException e) {
        System.out.println("Error de lectura escritura.");
    }
}
```

Es necesario retroceder el puntero sobre el último precio leído. Para ello se retroceden 8 posiciones correspondientes al número de bytes que ocupa un valor double.

Ejercicio 7.39:

Escriba un método, de nombre mostrarArchivoPrecios, que reciba por parámetro el nombre de un archivo que almacena una serie de referencias y precios de artículos. El método leerá los valores del archivo y los mostrará por pantalla.

Planteamiento: En primer lugar se crea un flujo para la lectura del archivo mediante un objeto de la clase `RandomAccessFile`. Mediante un bucle se van leyendo alternativamente del archivo los valores correspondientes a referencias y a artículos que son mostrados por pantalla. Finalmente se cierra el flujo sobre el archivo.

Parámetros: Nombre del archivo cuyos valores correspondientes a referencias y precios se mostrarán por pantalla.

Valor de retorno: El método no devuelve ningún valor de retorno.

Solución:

```
public void mostrarArchivoPrecios(String nombreArchivo) throws IOException{
   int referencia;
   double precio;
   boolean finArchivo = false;
   RandomAccessFile archivo = null;
   archivo = new RandomAccessFile(nombreArchivo, "rw");
   do {
      try {
         referencia = archivo.readInt();
         System.out.println("Referencia:  "+ referencia);
         precio = archivo.readDouble();
         System.out.println("Precio:  "+ precio);
      } catch (EOFException e) {
         finArchivo = true;
         archivo.close();
      }
   } while (!finArchivo);
}
```

Se va leyendo cada referencia y su correspondiente precio.

Ejercicio 7.40:

Escriba un programa que verifique el funcionamiento de los métodos de los tres ejercicios anteriores.

Planteamiento: En primer lugar se añaden los métodos a una clase de nombre EjFlujosAccesoAleatorio. El programa crea dos arrays para almacenar, respectivamente, los valores correspondientes. A continuación se instancia un objeto de la clase para invocar sobre él los métodos. Las excepciones que puedan derivarse de los métodos son capturadas y su tratamiento se limita a mostrar el correspondiente mensaje por pantalla.

Solución:

```
public static void main(String[] args) {
   int[] referencias = {10, 20, 30, 40, 50};
   double[] precios = {100.0, 50.0, 100.0, 50.0, 100.0};
   EjFlujosAccesoAleatorio ej = new EjFlujosAccesoAleatorio();
   try{
      ej.obtenerArchivoPrecios(referencias,
                               precios,
                               "C:/misArchivos/articulos.txt");
      ej.mostrarArchivoPrecios("C:/misArchivos/articulos.txt");
      ej.actualizarArchivoPrecios("C:/misArchivos/articulos.txt");
      ej.mostrarArchivoPrecios("C:/misArchivos/articulos.txt");
   }catch(IOException ioe){
      System.out.println("Error de lectura Escritura: "+ ioe.toString());
   }
}
```

LECTURA Y ESCRITURA DE OBJETOS

Ejercicio 7.41:

Considere la clase Punto que se presenta a continuación.

```
public class Punto {
```

```
    private int coordX;
    private int coordY;
    private double distOrigen;
    private String color;

    public Punto(int cx, int cy, String colorp) {
        coordX = cx;
        coordY = cy;
        distOrigen = Math.sqrt(cx*cx + cy*cy);
        color = colorp;
    }

    public int obtenerCoordX(){
        return coordX;
    }

    public int obtenerCoordY(){
        return coordY;
    }

    public double obtenerDistOrigen(){
        return distOrigen;
    }

    public String obtenerColor(){
        return color;
    }
    public String toString(){
        String resul = "";
        resul += "(" + coordX + ", " + coordY +")" + "Dist: " + distOrigen + "color: " + color;
        return resul;
    }
}
```

Se pide escribir una clase, de nombre PruebaFlujos2, donde se defina un método estático de nombre almacenarColPuntosEnArchivo que reciba una colección de objetos de la clase Punto. El método también recibirá por parámetro el nombre de un archivo para volcar sobre él la información de los objetos Punto contenidos en la colección. El método no capturará ninguna excepción.

Planteamiento: En primer lugar, se crea un flujo de escritura a partir del nombre del archivo recibido por parámetro. Se procede a continuación a obtener un iterador para recorrer los elementos de la colección recibida mediante el correspondiente bucle. En cada iteración del bucle se almacenan en el archivo la coordenada según el eje de abscisas(x), la coordenada según el eje de ordenadas(y) y el color de cada objeto Punto de la colección círculo.

Valor de retorno: El método no devuelve ningún valor de retorno, por lo que se utiliza la cláusula `void`.

Solución:

```
public class PruebaFlujos2 {

    public PruebaFlujos2() {
    }
```

```
    public static void almacenarColPuntosEnArchivo(Collection<Punto> puntos,
                                              String nombreArchivo) throws IOException{
        PrintWriter archivoPuntos=new PrintWriter(
                                    new BufferedWriter(
                                        new FileWriter(nombreArchivo)));
        Iterator<Punto> puntosIt = puntos.iterator();
        while(puntosIt.hasNext()){
            Punto p=puntosIt.next();
            archivoPuntos.println(p.getCoordX());
            archivoPuntos.println(p.getCoordY());
            archivoPuntos.println(p.getColor());
        }
        archivoPuntos.close();
    }
```

Se obtiene un iterador para recorrer la colección.

Se escribe en el archivo la información de cada punto.

Ejercicio 7.42:

Añada a la clase anterior un método estático, de nombre obtenerArrayListDeArchivo, que reciba por parámetro el nombre de un archivo con los datos correspondientes a varios objetos de la clase Punto. El método construirá y devolverá un objeto ArrayList<Punto> con los datos de los puntos leídos del archivo especificado. El método no capturará ninguna excepción.

Planteamiento: El método obtiene un flujo de lectura al archivo especificado a través de un objeto de la clase Scanner. A continuación se crea un nuevo objeto de la clase ArrayList para que almacene objetos de tipo Punto.

Para cada objeto Punto se leen del archivo los tres datos necesarios para invocar al constructor de la clase (coordenada x, coordenada y, color). Una vez creado el objeto de la clase Punto es añadido al ArrayList mediante el método add(). Finalmente el objeto ArrayList es devuelto.

Parámetros: Nombre del archivo con la información de los objetos Punto.

Valor de retorno: Objeto de la clase ArrayList con los objetos de la clase Punto construidos a partir de la información contenida en el archivo especificado.

Solución:

```
public static ArrayList<Punto> obtenerArrayListDeArchivo(String nombreArchivo)
                              throws IOException, FileNotFoundException{
    Scanner archivoPuntos = new Scanner(new File(nombreArchivo));
    ArrayList<Punto> listaPuntos = new ArrayList<Punto>();
    while( archivoPuntos.hasNextInt()){
        Punto p = new Punto(archivoPuntos.nextInt(),
                            archivoPuntos.nextInt(),
                            archivoPuntos.next());
        listaPuntos.add(p);
    }
    archivoPuntos.close();
    return listaPuntos;
}
```

Se crea un objeto Scanner para leer la información del archivo.

Se construye cada objeto Punto a partir de la información del archivo.

Ejercicio 7.43:

Añada a la clase anterior un método main() para comprobar el funcionamiento de los métodos estáticos de los dos ejercicios anteriores. Se crearán varios objetos Punto que se almacenarán en un array. A partir del

array se obtendrá una colección para guardar sus elementos en un archivo. Para finalizar se obtendrá un objeto ArrayList<Punto> con los objetos Punto creados a partir de los datos almacenados en el archivo.

Planteamiento: Se implementa lo indicado en el enunciado en ese orden.

Solución:
```
public static void main(String[] args) {
    Punto p1 = new Punto(1, 0, "Blanco");
    Punto p2 = new Punto(2, 0, "Rojo");
    Punto p3 = new Punto(0, 3, "Azul");
    Punto p4 = new Punto(0, 4, "Verde");
    Punto p5 = new Punto(5, 5, "Gris");

    Punto[] arrPuntos = {p1, p2, p3, p4, p5};
    try{
        PruebaFlujos2.almacenarColPuntosEnArchivo((Arrays.asList(arrPuntos)),
                                           "C:/misArchivos/arPuntos.txt");

        ArrayList<Punto> lp =
            new PruebaFlujos2.obtenerArrayListDeArchivo("C:/misArchivos/arPuntos.txt");
        System.out.println(lp);
    }catch (IOException ioE){
        System.out.println("Error: " + ioE.toString());
    }
}
```

Objetos de la clase Punto.

Se obtiene una lista con los objetos de la clase Punto a partir de la información del archivo.

Ejercicio 7.44:

Escriba un método, de nombre almacenarColPuntosComoObjetos, que reciba por parámetro una colección de objetos Punto y el nombre del archivo donde se almacenarán. El método escribirá directamente cada objeto Punto al archivo.

Planteamiento: En primer lugar es necesario hacer a la clase `Punto` serializable, para ello basta con hacer que implemente la interfaz `Serializable`. En el método se abre un flujo de escritura de objetos representado por un objeto de la clase `ObjectOutputStream`. Se obtiene un iterador para leer los objetos de la colección recibida y escribirlos al archivo con el método `writeObjet()`.

Parámetros: Colección de objetos de la clase `Punto` y cadena de caracteres con el nombre del archivo donde los objetos de la colección serán escritos.

Valor de retorno: El método no devuelve ningún valor de retorno.

Solución:

Se hace la clase `Punto` seralizable:

```
public class Punto implements Serializable {. . .}
```

El método solicitado quedaría como sigue:

```
public void almacenarColPuntosComoObjetos(Collection<Punto> puntos, String
                                    nombreArchivo) throws IOException{
    ObjectOutputStream archivoPuntos = new ObjectOutputStream(
                                    new FileOutputStream(nombreArchivo));
```

```
        Iterator<Punto> puntosIt = puntos.iterator();
        while(puntosIt.hasNext()){
            Punto p=puntosIt.next();
            archivoPuntos.writeObject(p); <-----
        }
        archivoPuntos.close();
    }
```

Los objetos se escriben directamente en el archivo con el método `writeObject()`.

Ejercicio 7.45:

Escriba un método, de nombre obtenerArrayListDeArchivoComoObjeto, que reciba por parámetro el nombre de un archivo donde se encuentran almacenados varios objetos de la clase Punto. El método leerá los objetos del archivo y los almacenará en un ArrayList que, finalmente, será devuelto.

Planteamiento: Se abre un flujo de lectura de objetos a partir del nombre de archivo recibido por parámetro. Se construye un `ArrayList` para que almacene referencias a objetos de la clase `Punto`. Se procede, mediante un bucle, a la lectura de todos los objetos almacenados en el archivo. Para leer cada objeto se utiliza el método `readObject()`. Una vez leído, el objeto es añadido a la lista mediante el método `add()`. Se utiliza la captura de la excepción `EOFException` para determinar el fin del archivo. Finalmente la lista obtenida es devuelta.

Solución:

```
public ArrayList<Punto> obtenerArrayListDeArchivoComoObjeto(String nombreArchivo)
                                    throws ClassNotFoundException, IOException{
    ObjectInputStream archivoPuntos = new ObjectInputStream(
                                    new FileInputStream(nombreArchivo));
    ArrayList<Punto> listaPuntos = new ArrayList<Punto>();
    try{
        do{
            Punto p = (Punto)archivoPuntos.readObject(); <-----
            listaPuntos.add(p);
        }while(true);
    }catch(EOFException eofE){} <-----
    return listaPuntos;
}
```

Los objetos se leen directamente con el método `readObject()`.

Se utiliza el lanzamiento de la excepción `EOFException` para determinar el fin del archivo.

Ejercicio 7.46:

Escriba un método, de nombre escribirArrayListEnArchivo, que reciba por parámetro un objeto ArrayList de objetos Punto, así como el nombre del archivo donde se almacenará. El método almacenará directamente el objeto ArrayList sobre el archivo recibido. No se capturará ninguna excepción que pueda producirse.

Planteamiento: Se abre un flujo para la escritura de objetos a partir del nombre de archivo recibido por parámetro. Dado que la clase `ArrayList` implementa la interfaz `Serializable`, y para la resolución de los ejercicios anteriores se ha hecho que la clase `Punto` también lo haga, se puede escribir directamente el objeto `ArrayList<Punto>` recibido al archivo mediante el método `writeObject()`. Finalmente se procede a cerrar el flujo de escritura sobre el archivo.

Solución:

```
public void escribirArrayListEnArchivo(ArrayList<Punto> puntos, String nombreArchivo)
                                    throws IOException{
    ObjectOutputStream archive = new ObjectOutputStream(new FileOutputStream(nombreArchivo));
    archivo.writeObject(puntos); <-----
    archivo.close();
}
```

Se escribe directamente el objeto `ArrayList` en el archivo.

Ejercicio 7.47:

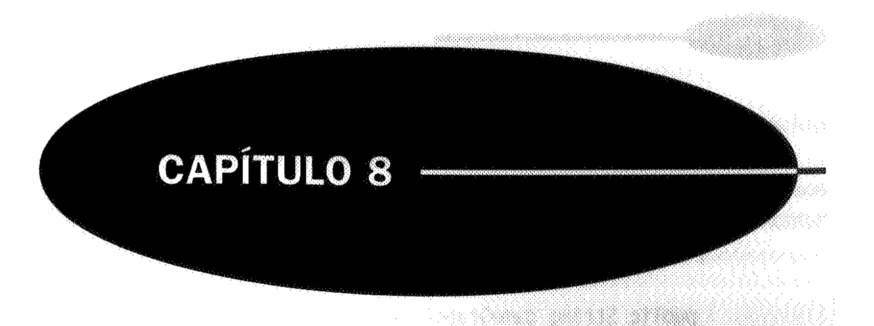
Escriba un método, de nombre leerArrayListDeArchivo, que reciba por parámetro el nombre de un archivo en el que se ha escrito previamente un objeto ArrayList que almacenaba objetos de la clase Punto. El método leerá el objeto ArrayList del archivo y lo devolverá. No se capturará ninguna excepción que pueda producirse.

Planteamiento: Se abre un flujo para la lectura de objetos a partir del nombre de archivo recibido por parámetro. Con ayuda del método `readObject()` se procede a leer el objeto `ArrayList` almacenado previamente en el archivo, para, a continuación, devolverlo.

Solución:

```
public ArrayList<Punto> leerArrayListDeArchivo(String nombreArchivo)
                                        throws IOException, ClassNotFoundException{
   ObjectInputStream archivo= new ObjectInputStream(
                                              new FileInputStream(nombreArchivo));
   ArrayList<Punto> al=(ArrayList<Punto>)(archivo.readObject());
   archivo.close();
   return al;
}
```

Lectura del objeto `ArrayList` con el método `readObject()`.



Interfaces

8.1 DEFINICIÓN Y USO DE INTERFACES

Una interfaz es una declaración de métodos no estáticos y campos estáticos finales cuyo fin es definir la estructura y comportamiento de las clases que la implementen de forma que presenten la misma interfaz pública.

Los elementos de una interfaz son públicos por definición: no es necesario poner la palabra `public`, ni se pueden cambiar sus derechos de acceso.

Los atributos siempre son finales, sin necesidad del calificativo `final`, y hay que darles un valor inicial.

Dada una interfaz, cualquier clase puede presentar dicha interfaz mediante un mecanismo denominado implementación de interfaces. Para ello se escribe:

```
class NombreClase implements NombreInterfaz {...}
```

Cuando una clase implemente una interfaz, tendrá que sobrecargar sus métodos con acceso público. Con otro tipo de acceso, el compilador lanzará un error.

Como ejemplo se va a utilizar el caso de una aplicación universitaria. En ella serán docentes los alumnos y los profesores, pero no los bedeles (ni otros empleados). Los docentes se distinguen de otras personas en que tendrán un grupo y un horario.

Para ello se va a definir una interfaz de nombre `Docente` con las operaciones `ponGrupo()`, `dameGrupo()` y `dameHorario()`. Para los no docentes estas operaciones no tienen sentido. Sin embargo, sí es cierto que todas los alumnos, profesores, bedeles, etc, son personas (es decir se pueden crear como derivadas de la clase `Persona`). De esta forma se puede escribir:

```
interface Docente {
   void    ponGrupo(String grupo, Horario horario);

   String  dameGrupo();
   Horario dameHorario();
}

class Alumno extends Persona implements Docente {
   private String  grupo;
```

Alumno no solo va a implementar la interfaz Docente, también extiende Persona.

```
    private Horario horario;
    public void ponGrupo(String grupo, Horario horario) {
        this.grupo   = grupo;
        this.horario = horario;
    }

    public String dameGrupo() {
        return grupo;
    }

    public Horario dameHorario() {
        return horario;
    }
}

abstract class Empleado extends Persona { ... }

class Profesor extends Empleado implements Docente {
    String grupo;
    boolean esMatutino;

    public void ponGrupo(String grupo, Horario horario) {
        this.grupo  = grupo;
        esMatutino = (horario == Horario.MAÑANA);
    }

    public String dameGrupo() {
        return grupo;
    }

    public Horario dameHorario() {
        if (esMatutino){
            return MAÑANA;
        }else{
            return TARDE;
        }
    }
}

// Bedel no implementa Docente
class Bedel extends Empleado { ... }
```

Un Profesor es una Persona Empleada que además implementa la interfaz Docente.

Un Bedel es una Persona Empleada pero no implementa la interfaz Docente.

Fíjese cómo el hecho de implementar una interfaz no obliga a la forma en que se sobreescriba el código de los métodos pertenecientes a la misma. En el ejemplo se ha hecho de forma diferente intencionadamente para demostrar esta diferencia. Una clase puede implementar varias interfaces, con la siguiente sintaxis:

```
class ClaseA implements InterfaceB, InterfaceC { ... }
```

La línea anterior declara una clase ClaseA que implementa las interfaces InterfaceB e InterfaceC.

Si alguno de los métodos queda sin implementar, la clase es abstracta. En ese caso, es necesario definirlo de forma explícita, añadiendo el calificativo abstract a la clase y añadiendo la cabecera del método que se deja sin implementar

anteponiendo a su vez el calificativo abstract. De esta forma queda patente que se está definiendo una implementación parcial, con la intención de que al programador no se le pasen cosas desapercibidas.

Una interfaz puede heredar de tantas interfaces como sea necesario. Lo que se consigue es añadir nuevos métodos a los existentes en las interfaces de las que hereda. Una clase que implemente esta interfaz, tiene que implementar todos los métodos declarados en ella, además de los heredados de las otras.

```
interface InterfazNueva extends InterfaceX, InterfaceY { ... }
```



Problemas resueltos

INTERFACES

Ejercicio 8.1:

Escriba una interfaz ColeccionSimple que declare los siguientes métodos:

- *estáVacía(): devuelve true si la colección está vacía y false en caso contrario.*
- *extraer(): devuelve y elimina el primer elemento de la colección.*
- *primero(): devuelva el primer elemento de la colección.*
- *añadir(): añade un objeto por el extremo que corresponda y devuelve true si se ha añadido y false en caso contrario.*

A continuación, escriba una clase PilaArray que implemente esta interfaz utilizando para ello un array de Object y un contador de objetos.

Planteamiento: En la interfaz se declaran todos los métodos sin implementar ninguno.

Para la clase se utilizan como atributos un `array` de `Object` y un entero que sirve de contador de objetos.

El constructor recibe por parámetro el tamaño máximo de la pila.

El método `estáVacía()` comprueba si el contador es 0.

El método `añadir()` comprueba que cabe el elemento y si es así, lo añade en la celda que indica el contador, posteriormente incrementa el valor del contador. Devuelve `true` o `false` según si se ha añadido o no.

`primero()` si está vacía lanza `NoSuchElementException`, si no devuelve el elemento que está en la celda indicada por `contador - 1`.

`extraer()` si está vacía lanza `NoSuchElementException`, si no decrementa el contador y devuelve el elemento que está en la celda indicada por el contador después de decrementarse.

`NoSuchElementException` hereda de `RuntimeException`, por lo que no necesita ser declarada en la cláusula `throws` del método.

Se sobreescribe el método `toString()` de manera que devuelva los objetos encerrados entre corchetes y separados por comas, comenzando por el último elemento insertado y terminando por el primero.

Solución:

```java
public interface ColeccionSimple {
    boolean estáVacía();
    boolean añadir(Object o);
    Object primero();
    Object extraer();
}
import java.util.NoSuchElementException;
public class PilaArray implements ColeccionSimple{

    private Object[] array;
    private int contador;

    public PilaArray(int tamañoMáximo) {
        array    = new Object[tamañoMáximo];
        contador = 0;
    }
    public boolean estáVacía(){
```

> No se implementa ningún método.

> Los métodos son `public` aunque no se indique explícitamente.

> `PilaArray` implementa `ColeccionSimple`.

> `Array` de `Object`, podrá referenciar a cualquier tipo de objetos.

```
            return contador == 0;
        }

        public boolean añadir(Object o){
            if (contador == array.length)
                return false;
            else{
                array[contador] = o;
                contador++;
                return true;
            }
        }

        public Object primero(){
            if (estáVacía())
                throw new NoSuchElementException(); 
            else
                return array[contador-1];
        }

        public Object extraer(){
            if (estáVacía())
                throw new NoSuchElementException();
            else
                return array[--contador];
        }

        public String toString(){
            String s = "[";
            for (int i = contador - 1; i >= 0; i—){
                s += array[i].toString()+",";
            }
            if (!estáVacía())
                return s.substring(0, s.length() - 1) + "]";
            else
                return s + "]";
        }
    }
```

No es necesario declarar en la cabecera la excepción NoSuchElementException.

Se predecrementa el contador.

Se elimina la última coma.

Comentario: En el método toString(), se podría declarar la variable s del tipo StringBuffer para conseguir mayor eficiencia.

Ejercicio 8.2:

Escriba una clase, de nombre PruebaColeccionSimple, en la que se implementen dos métodos:

a) rellenar(): recibe por parámetro un objeto de tipo ColeccionSimple y añade los números del uno al diez.
b) imprimirYVaciar(): recibe por parámetro un objeto de tipo ColeccionSimple y va extrayendo e imprimiendo los datos de la colección hasta que se quede vacía.
c) En la aplicación principal cree un objeto de tipo PilaArray, llame a los métodos rellenar() e imprimirYVaciar() pasando este objeto por parámetro. Escriba en la pantalla el contenido de la pila antes y después de llamar a imprimirYVaciar().

Planteamiento: En los métodos rellenar() e imprimirYVaciar()el parámetro puede ser un objeto declarado del tipo de una interfaz. Lo importante, y con lo que cuenta el método, es que reciba un objeto de una clase que implemente esta interfaz y, por tanto, tendrá implementados los métodos de ColeccionSimple.

a) rellenar(): se realiza un for con un int que vaya de uno a diez y se llama a añadir() pasando este entero por parámetro. Se realiza una conversión automática de int en Integer.
b) imprimirYVaciar(): se realiza un while, y mientras que no esté vacía se imprime en la pantalla el resultado del método extraer().
c) Se crea el objeto de tipo PilaArray, al pasarlo por parámetro a los métodos que reciben ColeccionSimple no hay problemas de tipo. Como Pila hereda de ColeccionSimple, la conversión de tipo hacia arriba se realiza automáticamente. imprimirYVaciar() imprimirá en el contrario al que se han añadido.

Solución:

```
public class PruebaColeccionSimple {

    public static void rellenar(ColeccionSimple c){
        for (int i = 1; i <= 10; i++){
            c.añadir(i); <-----
        }
    }

    public static void imprimirYVaciar(ColeccionSimple colección){
        while(!colección.estáVacía()){
            System.out.println(colección.extraer()); <-----
        }
    }

    public static void main(String[] args) {
        PilaArray p = new PilaArray(20);
        rellenar(p); <-----
        System.out.println("La pila: "+p);
        imprimirYVaciar(p);
        System.out.println("La pila: "+p);
    }
}
```

Por ligadura dinámica se llama al método añadir() que corresponda a c.

Se llama automáticamente al método toString() del objeto que devuelve.

p se convierte a ColeccionSimple automáticamente.

Nota: Fíjese que los métodos rellenar() e imprimirYVaciar() son exactamente iguales que en el Ejercicio 5.21 del Capítulo 5. Cuando se recibe un parámetro, no importa si el tipo es una clase o una interfaz, lo importante es que con el objeto recibido se pueda llamar a los métodos. Por ligadura dinámica se llamará al método correspondiente a la clase del objeto que se reciba por parámetro.

Ejercicio 8.3:

Escriba una clase Cola que implemente la interfaz ColeccionSimple usando un objeto de la clase LinkedList.

Planteamiento: Cola implementa la interfaz ColeccionSimple, para ello delegará en un atributo de tipo LinkedList. Se establece una relación de composición entre Cola y LinkedList. Para implementar la clase Cola, se inserta por el final de la LinkedList y se extrae por el principio.

Solución:

Cola implementa ColeccionSimple.

```
import java.util.LinkedList;
public class Cola implements ColeccionSimple{ <-----
```

```
    private LinkedList<Object> lista;

    public Cola() {
        lista = new LinkedList<Object>();
    }

    public boolean añadir(Object obj){
        lista.addLast(obj);
        return true;
    }

    public Object extraer(){
        return lista.removeFirst();
    }

    public boolean estáVacía(){
        return lista.isEmpty();
    }

    public Object primero(){
        return lista.getFirst();
    }

    public String toString(){
        return lista.toString();
    }
}
```

Atributo LinkedList para elementos Object.

Siempre devuelve true, ya que siempre se añade el elemento.

Comentarios: En los métodos primero() o extraer(), los métodos de LinkedList lanzan la excepción NoSuchElementException si la cola estuviese vacía.

Ejercicio 8.4:

Escriba una clase PilaArrayList que implemente la interfaz ColeccionSimple heredando de ArrayList.

Planteamiento: Java permite heredar de una sola clase e implementar todas las interfaces que se desee. Por tanto, para este ejercicio, la clase PilaArrayList hereda de ArrayList e implementa la interfaz ColeccionSimple. Para implementar los métodos de ColeccionSimple se utilizan los métodos heredados de ArrayList. En esta implementación se inserta y se extrae por el principio del ArrayList.

Solución:

```
import java.util.ArrayList;
public class PilaArrayList extends ArrayList<Object> implements ColeccionSimple{

    public PilaArrayList() {
    }

    public boolean estáVacía(){
        return isEmpty();
    }

    public boolean añadir(Object obj){
```

Hereda de ArrayList de Object e implementa ColeccionSimple al mismo tiempo.

Llama a super() automáticamente. En este caso al constructor de ArrayList.

```
        add(0, obj);
        return true;
    }

    public Object primero(){
        return get(0);
    }

    public Object extraer(){
        return remove(0);
    }
}
```

Se invocan los métodos de ArrayList como si estuviesen en esta clase, ya que se heredan.

Nota: El método toString() no se implementa porque se hereda de ArrayList.

Comentario: La interfaz ColeccionSimple y sus descendientes se han implementado para almacenar objetos de tipo Object. Es preferible declarar estas clases genéricas con tipo parametrizado, de manera que el cliente elija qué tipo de datos quiere almacenar. Para más información acerca de cómo hacer la clase ColeccionSimple genérica consulte el Ejercicio 9.6 del Capítulo 9.

En los métodos primero() o extraer(), los métodos de ArrayList lanzan la excepción IndexOutOfBounds Exception si la pila estuviese vacía.

Ejercicio 8.5:

Modifique el método main de la clase del Ejercicio 8.2 para que en lugar de crear un objeto de tipo PilaArray, se creen dos objetos de tipo Cola y PilaArrayList. Realice las mismas operaciones que realizó para el objeto PilaArray.

Planteamiento: Es el mismo que en el Ejercicio 8.2, los métodos rellenar() e imprimirYVaciar() sirven para objetos de cualquier clase que implemente ColeccionSimple, por tanto el código será el mismo excepto por la creación de los objetos.

Solución:

```
public class Main {
    public static void rellenar(ColeccionSimple colección){
        for (int i = 1; i <= 10; i++){
            colección.añadir(i);
        }
    }

    public static void imprimirYVaciar(ColeccionSimple colección){
        while(!colección.estáVacía()){
            System.out.println(colección.extraer());
        }
    }
    public static void main(String[] args) {
        Cola cola = new Cola();
        rellenar(cola);
        System.out.println("La cola: " + cola);
        imprimirYVaciar(cola);
        System.out.println("La cola: " + cola);
        PilaArrayList pila = new PilaArrayList();
```

Son los mismos métodos que en el Ejercicio 8.2.

Por ligadura dinámica se ejecutan los métodos que corresponden a la clase de c.

Se hace automáticamente la conversión a ColeccionSimple.

```
            rellenar(pila);
            System.out.println("La pila: " + pila);
            imprimirYVaciar(pila);
            System.out.println("La pila: " + pila);
        }
    }
```

Ejercicio 8.6:

Escriba un programa para una biblioteca que contenga libros y revistas.

a) Las características comunes que se almacenan tanto para las revistas como para los libros son el código, el título y el año de publicación. Estas tres características se pasan por parámetro en el momento de crear los objetos.

b) Los libros tienen además un atributo prestado. Los libros cuando se crean no están prestados.

c) Las revistas tienen un número. En el momento de crear las revistas se pasa el número por parámetro.

d) Tanto las revistas como los libros deben tener (aparte de los constructores) un método toString() que devuelve el valor de todos los atributos en una cadena de caracteres. También tienen un método que devuelve el año de publicación y otro para el código.

e) Para prevenir posibles cambios en el programa se tiene que implementar una interfaz Prestable con los métodos prestar(), devolver() y prestado(). La clase Libro implementa esta interfaz.

Planteamiento:

a) Se implementa una superclase de `Libro` y `Revista` con sus características comunes, que se llama `Publicación`. En esta clase además de declarar los tres atributos, se implementa un constructor que reciba por parámetro el valor de los tres atributos. También se implementan los métodos `getAño()`, `getCódigo()` y un método `toString()` que devuelve la información de estos tres atributos en forma de cadena de texto.

b) Se implementan las clases `Libro` y `Revista` que añaden sus nuevos atributos.

c) Se escriben sus constructores, que llaman al constructor de la superclase,

d) Se sobreescribe el método `toString()` que también llama al método `toString()` de la superclase.

e) La interfaz `Prestable` declara los métodos indicados sin implementarlos, la clase `Libro` implementa `Prestable` y, por tanto, todos sus métodos.

Solución:

```
public class Publicacion {

    private String código;
    private String título;
    private int año;

    public Publicacion(String código, String título, int año){
        this.código = código;
        this.título = título;
        this.año    = año;
    }

    public int getAño(){
        return año;
    }

    public String getCódigo(){
        return código;
    }
```

```java
    public String toString(){
        return "Código: " + código +
              "\nTítulo: " + título +
              "\nAño de publicación: " + año + "\n";
    }
}

public class Revista extends Publicacion {

    private int número;

    public Revista(String código, String título, int año, int número){
        super(código, título, año);
        this.número = número;
    }

    public String toString(){
        return super.toString() + "Número: " + número + "\n";
    }
}
```

Llamada al constructor de la superclase.

Llamada al método toString() de la superclase.

Llamada al método toString() de la superclase.

```java
public interface Prestable {
    void prestar();
    void devolver();
    boolean prestado();
}

public class Libro extends Publicacion implements Prestable{
    private boolean prestado;

    public Libro(String código, String título, int año){
        super(código, título, año);
        prestado = false;
    }

    public void prestar(){
        prestado = true;
    }

    public void devolver(){
        prestado = false;
    }

    public boolean prestado(){
        return prestado;
    }

    public String toString(){
        return super.toString() + (prestado ? "prestado" : "no prestado") + "\n";
    }
}
```

Ejercicio 8.7:

Escriba una aplicación en la que se implementen dos métodos:

a) cuentaPrestados(): recibe por parámetro un array de objetos y devuelve cuántos de ellos están prestados.
b) publicacionesAnterioresA(): recibe por parámetro un array de Publicaciones y un año, devuelve cuántas publicaciones tienen fecha anterior al año recibido por parámetro.
c) En el método main() crear un array de Publicaciones con 2 libros y 2 revistas, prestar uno de los libros, mostrar por pantalla los datos almacenados en el array y mostrar por pantalla cuántas publicaciones hay prestadas y cuántas hay anteriores a 1980 utilizando los métodos escritos anteriormente.

Planteamiento:

a) cuentaPrestados() recibe un array de Objetos para que se pueda utilizar tanto con Publicaciones como con cualquier otro tipo de objetos que se puedan prestar. Para contar cuántos objetos están prestados, se utiliza un contador que se inicializa a 0. Se recorre el array comprobando si cada una de las celdas es Prestable. Si lo es, se comprobará si está prestado y en este caso se incrementará un contador. Para comprobar si está prestado, antes hay que convertirlo a Prestable, ya que la clase Object no tiene el método prestado().
b) publicacionesAnterioresA() utiliza también un contador inicializado a 0. Se recorre el array obteniendo el año de cada una de las publicaciones, se comprueba si es menor que el año recibido por parámetro y si es así, se incrementa el contador.
c) El array se puede crear de varias maneras, en esta ocasión se le dan los datos en el momento de crearlo. Para prestar uno de los libros, se accede a una de las celdas, se transforma a Libro y se llama al método prestar().

Para mostrar por pantalla la información se recorre el array y se muestra cada uno de los objetos utilizando su método toString(), al que no es necesario llamar explícitamente. Después se llama a los métodos implementados anteriormente.

Solución:

```
public class Main {

    public static int cuentaPrestados(Object[] lista){
        int contador = 0;
        for (Object o: lista){
            if (o instanceof Prestable && ((Prestable)o).prestado())
                contador++;
        }
        return contador;
    }

    public static int publicacionesAnterioresA(Publicacion[] lista, int año){
        int contador = 0;
        for (Publicacion p: lista){
            if (p.getAño()<año)
                contador++;
        }
        return contador;
    }
    public static void main(String[] args) {
        Publicacion[] biblioteca = {new Libro("CC1", "La fundación", 1951),
                                    new Revista("CR1", "El jueves", 2002, 1305),
                                    new Libro("CC2", "El neuromante", 1984),
                                    new Revista("DR1", "Quo", 2002, 81)};
```

Se comprueba que es Prestable.

Se transforma a Prestable para llamar a prestado().

Todas las clases que hereden de Publicación tienen el método getAño().

```
        Libro l = (Libro)biblioteca[0];
        l.prestar();
        for(Publicacion p: biblioteca){
            System.out.println(p);
        }
        System.out.println(publicacionesAnterioresA(biblioteca, 1980)+
                           " publicaciones anteriores a 1980");
        System.out.println(cuentaPrestados(biblioteca)+
                           " libros prestados");
    }
}
```

Se transforma a `Libro` para poder llamar a `prestar()`.

Ejercicio 8.8:

Escriba una clase DiscoPrestable que herede de la clase Disco, escrita en el Ejercicio 5.13 del Capítulo 5, sobre herencia e implemente la interfaz Prestable.

Planteamiento: La clase `DiscoPrestable` hereda de `Disco` e implementa el interfaz `Prestable` implementando todos sus métodos. Para implementar los métodos de `Prestable` declara un atributo `boolean` prestado.

Solución:

```
public class DiscoPrestable extends Disco implements Prestable{

    private boolean prestado;

    public DiscoPrestable(String título, String autor, Formato formato, double duración, Genero
género) {
        super(título, autor, formato, duración, género);
        prestado = false;
    }

    public void prestar(){
        prestado = true;
    }

    public void devolver(){
        prestado = false;
    }

    public boolean prestado(){
        return prestado;
    }

    public String toString(){
        return super.toString() + (prestado ? "\nprestado" : "\nno prestado") + "\n";
    }
}
```

Hereda de `Disco` e implementa `Prestable`.

Llamada al constructor de la superclase.

Implementación de los métodos de `Prestable`.

Comentario: Si la clase `Disco` estuviera en otro paquete, habría que importar del paquete la cláusula `import`.

Ejercicio 8.9:

En el Ejercicio 8.7 modifique el programa principal para que:

a) Cree un array de tres objetos Disco.
b) Introduzca un DiscoPrestable en cada celda.
c) Preste dos de ellos.
d) Calcule e imprima por pantalla cuántos están prestados utilizando el método cuentaPrestados().

Planteamiento:
a) Se declara el array de `Disco` y se crea con tamaño 3.
b) Se crean y se introducen los objetos `DiscoPrestable` celda a celda
c) Se prestan dos de ellos accediendo a la celda específica. Se necesita hacer una conversión de tipo a `DiscoPrestable` para llamar al método `prestar()`.
d) Se llama al método `cuentaPrestados()`, que al recibir un array de `Object`, acepta el array de `Disco`.

Solución:
```
public class Main {

    public static int cuentaPrestados(Object[] lista){ <- Igual que en el Ejercicio 8.7.
        int contador = 0;
        for (Object o: lista){
            if (o instanceof Prestable && ((Prestable)o).prestado())
                contador++;
        }
        return contador;
    }

    public static void main(String[] args) {

        Disco[] discos = new Disco[3];
        discos[0] = new DiscoPrestable("Hopes and Fears", "Keane",
                                Formato.mp3, 50, Genero.pop);
        discos[1] = new DiscoPrestable("How to dismantle an atomic bomb", "U2",
                                Formato.cdAudio, 60, Genero.rock);
        discos[2] = new DiscoPrestable("Soy gitano", "Camarón",
                                Formato.cdAudio, 40, Genero.flamenco);

        ((Prestable)discos[0]).prestar(); <- Conversión de tipo para poder llamar a prestar().
        ((Prestable)discos[2]).prestar();

        for (Disco d: discos){

            System.out.println(d);
        }
                                         Se convierte automáticamente a array de Object.
        System.out.println(cuentaPrestados(discos) + " discos prestados");

    }
}
```

Comentario: Fíjese en la utilidad de la interfaz `Prestable`, que ha permitido tratar por igual a `DiscoPrestable` que a `Libro`. El mismo método sirve para ambas clases, y serviría para cualquier clase que implemente `Prestable`. Al mismo tiempo `DiscoPrestable` sigue siendo un `Disco` y `Libro` sigue siendo una `Publicación`.

Ejercicio 8.10:

Escriba una clase Proyecto para almacenar información de los proyectos de fin de carrera de una titulación. Esta clase hereda de Publicación para reutilizar todo lo que ya está escrito, además añade un atributo que es el nombre de la carrera.

Los objetos de esta clase se podrán prestar, por lo que debe implementar Prestable.

Además, se debe almacenar en una estructura de datos que esté ordenada, por lo que debe implementar la interfaz Comparable de Java. El método compareTo() se debe implementar para que los proyectos se comparen por su código.

Por último, sobreescriba el método toString() para que se muestre la nueva información.

Planteamiento: La clase `Proyecto` hereda de `Publicación` e implementa `Prestable` y `Comparable`. Para el atributo `carrera`, se declara un tipo enumerado `Carrera` con los valores de las carreras que se necesiten. Para implementar la interfaz `Prestable` se añade un atributo `prestado` de tipo `boolean`. El método `compareTo()` se implementa para que compare por el código del proyecto de fin de carrera. El método `toString()` llama a `toString()` de la superclase y añade la información del atributo `carrera`.

Solución:

```
public enum Carrera {SIS, INF, TCO, SEI, ICA, DIN, IND}

public class Proyecto extends Publicacion implements Prestable, Comparable<Proyecto>{

    private Carrera carrera;
    private boolean prestado;

    public Proyecto(String código, String nombre, int año, Carrera carrera) {
        super(código, nombre, año);
        this.carrera = carrera;
    }

    public void prestar(){
        prestado = true;
    }

    public void devolver(){
        prestado = false;
    }

    public boolean prestado(){
        return prestado;
    }


    public int compareTo(Proyecto p){
        return getCódigo().compareTo(p.getCódigo());
    }

    public String toString(){
        return super.toString() + "Carrera: " + carrera + "\n";
    }

}
```

Si se implementa más de una interfaz se separan por comas

Comparable es genérico, hay que indicarle la clase.

La comparación entre `Proyectos` se hace por su código.

Ejercicio 8.11:

En el Ejercicio 8.7 modifique el programa principal para hacer lo siguiente:

a) *Cree un array de Publicaciones con tres celdas.*
b) *Introduzca un objeto de tipo Proyecto en cada celda.*
c) *Preste dos de ellos.*
d) *Calcule e imprima por pantalla cuántos están prestados utilizando el método cuentaPrestados()*
e) *Ordene el array.*
f) *Muestre por pantalla los objetos del array.*

Planteamiento:

a) Se declara el array de `Publicación` y se crea con tamaño 3.
b) Se crean y se introducen los objetos `Proyecto` celda a celda. No hay incompatibilidad de tipos, ya que los `Proyectos` son también `Publicaciones`.
c) Se prestan dos de ellos accediendo a la celda específica. Se necesita hacer una conversión de tipo a `Prestable` para llamar al método `prestar()`. No existe incompatibilidad de tipos en la conversión, ya que los Proyectos son también `Prestables`.
d) Se llama al método `cuentaPrestados()`, que al recibir un array de `Object`, acepta el array de `Publicación`.
e) Para ordenar el array se utiliza el método `Arrays.sort()` que recibe por parámetro un array de `Object`, pero especifica que los objetos almacenados deben implementar `Comparable`. En este caso se cumple que los proyectos son también `Comparable`.
f) Se muestra utilizando un `for` que recorra el array y vaya mostrando uno a uno todos los elementos del array. Aunque el array está declarado de `Publicación`, por ligadura dinámica se llamará al método `toString()` de `Proyecto`.

Solución:

```
import java.util.Arrays;
public class Main {

    public static int cuentaPrestados(Object[] lista){ ←  Igual que en el Ejercicio 8.7.
        int contador = 0;
        for (Object o: lista){
            if (o instanceof Prestable && ((Prestable)o).prestado())
                contador++;
        }
        return contador;
    }

    public static int publicacionesAnterioresA(Publicacion[] lista, int año){
        int contador = 0;
        for (Publicacion p: lista){
            if (p.getAño() < año)
                contador++;
        }
        return contador;

    }
    public static void main(String[] args) {
        Publicacion[] proyectos = new Publicacion[3];
        proyectos[0] = new Proyecto("SIS-04039","Editor de partituras musicales", 2005, Carrera.SIS);
        proyectos[1] = new Proyecto("INF-04003","Reconocimiento de voz", 2004, Carrera.INF);
```

Se convierten automáticamente a `Publicación`.

```
        proyectos[2] = new Proyecto("SIS-03014","Herramientas SAP", 2004, Carrera.SIS);
        ((Prestable)proyectos[0]).prestar();
        ((Prestable)proyectos[2]).prestar();

        System.out.println(cuentaPrestados(proyectos) + " proyectos prestados");

        Arrays.sort(proyectos);

        for (Publicacion p: proyectos){
            System.out.println(p);
        }
    }
}
```


Nota: Para más información acerca de `Arrays.sort` consulte el Capítulo 6.

Ejercicio 8.12:

Escriba una interfaz ListaString que herede de la interfaz Iterable<String> y añada los siguientes métodos:

- *getString(): devuelve el String almacenado en la posición que se pasa por parámetro.*
- *addString(): añade el String recibido por parámetro al final de la lista.*
- *size(): devuelve el número de elementos que hay en la lista.*
- *listIterator() devuelve un objeto de tipo ListIterator<String> para recorrer la lista completa.*
- *listIteratorComienzaPor(char c): devuelve un objeto de tipo ListIterator<String> para recorrer los elementos de la lista que comienzan por el carácter introducido por parámetro.*

Planteamiento: La interfaz `ListaString` hereda de `Iterable<String>`. Se hace con el objetivo de poder utilizar el `for` abreviado para recorrer una colección con objetos de `ListaString`.

La interfaz `Iterable` contiene el método `iterator()` que, lógicamente, no se implementa en `ListaString`, ya que es una interfaz. Este método se implementará en las clases que implementen `ListaString`. Los métodos `listIterator()` y `listIteratorComienzaPor(char c)` devuelven un `ListIterator` de `String`, eso se indica de la manera especificada en el enunciado del ejercicio. `ListIterator` igual que `Iterator` e `Iterable` son interfaces genéricas, por lo que hay que fijar el tipo actual por parámetro. Para más información sobre genericidad consulte el Capítulo 9.

Solución:

```
import java.util.ListIterator;

public interface ListaString extends Iterable<String>{
    String getString(int index);
    void addString(String s);
    int size();
    ListIterator<String> listIteratorComienzaPor(char c);
    ListIterator<String> listIterator();
}
```


Comentario: Para que el tipo de datos `ListaString` fuese completo necesitaría más métodos, como por ejemplo `eliminar()`. Por simplicidad no se implementan todos los métodos que corresponderían a este tipo de datos.

Ejercicio 8.13:

a) Escriba una clase ListaStringArray que implemente ListaString utilizando para ello un array de String y un contador de objetos.

Para implementar los métodos listIterator() y listIteratorComienzaPor() debe implementar dos clases (b)ListIteratorArray y (c)ListIteratorComienzaPor que implementen la interfaz ListIterator. No es necesario que implemente los métodos add(), set() y remove().

Planteamiento:

a) La clase `ListaStringArray` implementa `ListaString` y por tanto también implementa `Iterable<String>`. Es decir, además de implementar los métodos especificados en `ListaString`, debe implementar el método `iterator()`.

El método `iterator()` devuelve un objeto de la clase `Iterator`.

Para implementar `iterator()` se llama al método `listIterator()` que devuelve un `ListIterator<String>`. `ListIterator` hereda de `Iterator`, por lo que lo aceptará sin que haya incompatibilidad de tipos.

La interfaz `ListIterator` contiene los siguientes métodos:

- `add()`: opcional, no se pide que se implemente en el enunciado.
- `hasNext()`: devuelve `true` si existen más elementos por recorrer hacia adelante.
- `hasPrevious()`:devuelve `true` si existen elementos por recorrer hacia atrás.
- `next()`: devuelve el siguiente elemento de la lista.
- `nextIndex()`: devuelve la posición del siguiente elemento de la lista.
- `previous()`: devuelve el elemento anterior en la lista.
- `previousIndex()`: devuelve la posición del elemento anterior en la lista.
- `remove()`: opcional, no se pide que se implemente en el enunciado
- `set()`: opcional, no se pide que se implemente en el enunciado.

Para las operaciones opcionales, si no se implementan, su cuerpo consiste en lanzar la excepción `UnsupportedOperationException`.

b) La clase `ListIteratorArray` implementa `ListIterator<String>`, almacena una referencia a la lista que se está recorriendo y un índice de la posición en la que se encuentra.

En el constructor se recibe por parámetro la `ListaString` a recorrer.

Este índice se inicializa a –1 para que la primera llamada a `next()` devuelva el elemento en la posición 0.

Los métodos se implementan de la siguiente manera:

- `add()`: lanza `UnsupportedOperationException`.
- `hasNext()`: devuelve `true` si el índice es menor que el tamaño de la lista menos 1 (la lista comienza en la posición 0).
- `hasPrevious()`: devuelve `true` si el índice es mayor que 0.
- `next()`: incrementa el índice y devuelve el elemento al que referencia después de haber sido incrementado. En el caso de que no hubiese más elementos, se lanza `NoSuchElementException`.
- `nextIndex()`: devuelve el índice más 1.
- `previous()`: decrementa el índice y devuelve el elemento al que referencia después de haber sido decrementado. En el caso de que no hubiese más elementos, se lanza `NoSuchElementException`.
- `previousIndex()`: devuelve el índice menos 1.
- `remove()`: lanza `UnsupportedOperationException`.
- `set()`: lanza `UnsupportedOperationException`.

c) La clase `ListIteratorComienzaPor` implementa `ListIterator<String>`, almacena una referencia a la lista que está recorriendo, el carácter por el que deben comenzar las palabras recorridas, y un índice de la posición en la que se encuentra.

En el constructor se recibe por parámetro la `ListaString` a recorrer y el carácter por el que deben recorrer las palabras recorridas.

El índice se inicializa a –1 para que la primera llamada a `next()` devuelva el elemento en la posición 0 (si comenzase por el carácter debido).

Los métodos se implementan de la siguiente manera:

- add(): lanza UnsupportedOperationException.
- hasNext(): devuelve true si existe alguna cadena posterior que comience por el carácter adecuado. Para ello busca desde la posición indicada por el índice hacia adelante si alguna cadena de la lista cumple la condición.
- hasPrevious(): devuelve true si existe alguna cadena anterior que comience por el carácter adecuado. Para ello busca desde la posición indicada por el índice hacia atrás si alguna cadena de la lista cumple la condición.
- next(): devuelve la siguiente cadena que comience por el carácter adecuado y actualiza el índice. Para ello busca desde la posición indicada por el índice hacia adelante si alguna cadena de la lista cumple la condición, si no fuese así lanza NoSuchElementException.
- nextIndex(): devuelve la posición de la siguiente cadena que comience por el carácter adecuado. Para ello busca desde la posición indicada por el índice hacia adelante si alguna cadena de la lista cumple la condición, si no existe ninguna, devuelve la longitud de la lista.
- previous():devuelve la anterior cadena que comience por el carácter adecuado y actualiza el índice. Para ello busca desde la posición indicada por el índice hacia atrás si alguna cadena de la lista cumple la condición, si no fuese así lanza NoSuchElementException.
- previousIndex(): devuelve la posición de la anterior cadena que comience por el carácter adecuado. Para ello busca desde la posición indicada por el índice hacia atrás si alguna cadena de la lista cumple la condición, si no existe ninguna, devuelve –1.
- remove(): lanza UnsupportedOperationException.
- set(): lanza UnsupportedOperationException.

d) Una vez implementadas estas dos clases, se escribe la clase ListaStringArray que tiene como atributos un array de String y un contador.

El constructor recibe por parámetro el número máximo de cadenas que se pueden almacenar en la lista. Se crea un array con este tamaño y se inicializa el contador a 0.

En esta clase se implementan los siguientes métodos:

- getString(): devuelve el String almacenado en la posición que se pasa por parámetro.
- addString(): añade el String recibido por parámetro en la posición que indica el contador, e incrementa éste.
- size(): devuelve el valor almacenado en el contador.
- listIterator(): devuelve un objeto de tipo ListIterator<String> para recorrer la lista completa. Este objeto se crea dentro del método, como parámetro del constructor se pasa una referencia this, del propio objeto que ha llamado al método.
- listIteratorComienzaPor(char c): devuelve un objeto de tipo ListIterator<String> para recorrer la lista completa. Este objeto se crea dentro del método, como parámetro del constructor se pasa una referencia this, del propio objeto que ha llamado al método, y el carácter recibido por parámetro.
- iterator(): llama al método listIterator().

Solución:

```
import java.util.*;

public class ListIteratorArray implements ListIterator<String>{
    private ListaString l;
    private int index;

    public ListIteratorArray(ListaString lista) {
        l = lista;
```

Implementa ListIterator para String.

l de tipo ListaString para que sirva para otros tipos de ListaString.

```
            index = -1;
        }

        public void add(String s) {
            throw new UnsupportedOperationException();
        }
```

Los métodos add(), set() y remove() son opcionales, si no se implementan lanzan esta excepción.

```
        public boolean hasNext() {
            return index < l.size() - 1;
        }

        public boolean hasPrevious() {
            return index > 0;
        }

        public String next() {
            if(index == l.size() - 1)
                throw new NoSuchElementException();
            return l.getString(++index);
        }
```

En next() y previous() si no quedan elementos se lanza esta excepción.

```
        public int nextIndex() {
            return index + 1;
        }

        public String previous() {
            if(index == 0)
                throw new NoSuchElementException();
            return l.getString(--index);
        }

        public int previousIndex() {
            return index - 1;
        }

        public void remove() {
            throw new UnsupportedOperationException();
        }
```

Los métodos add(), set() y remove() son opcionales, si no se implementan lanzan esta excepción.

```
        public void set(String s) {
            throw new UnsupportedOperationException();
        }
    }

    import java.util.*;
    public class ListIteratorComienzaPor implements ListIterator<String>{
        private ListaString l;
        private int index;
        private char inicial;

        public ListIteratorComienzaPor(ListaString lista, char ini) {
```

```
        inicial = ini;
        l = lista;
        index = -1;
    }

    public void add(String s) {
        throw new UnsupportedOperationException();
    }

    public boolean hasNext() {
        int i;
        for (i = index + 1; i < l.size(); i++){
            if (l.getString(i).charAt(0) == inicial){
                return true;
            }
        }
        return false;
    }
```

Se inicializa a `index + 1` para que comience a buscar a partir del siguiente.

```
    public boolean hasPrevious() {
        int i;
        for (i = index - 1; i >= 0; i--){
            if (l.getString(i).charAt(0) == inicial){
                return true;
            }
        }
        return false;
    }

    public String next() {
        int i;
        for (i = index + 1; i < l.size(); i++){
            if (l.getString(i).charAt(0) == inicial){
                index = i;
                return l.getString(i);
            }
        }
        throw new NoSuchElementException();
    }
```

Se actualiza `index` y se devuelve la cadena en esa posición.

Si no queda ninguno que cumpla la condición, se lanza esta excepción.

```
    public int nextIndex() {
        int i;
        for (i = index + 1; i < l.size(); i++){
            if (l.getString(i).charAt(0) == inicial){
                return i;
            }
        }
        return l.size();
    }
```

Si no queda ninguno que cumpla la condición, se devuelve el tamaño de la lista.

```
    public String previous() {
```

```
        int i;
        for (i = index - 1; i >= 0; i—){
            if (l.getString(i).charAt(0) == inicial){
                index = i;
                return l.getString(i);
            }
        }
        throw new NoSuchElementException();
    }

    public int previousIndex() {
        int i;
        for (i = index - 1; i >= 0; i—){
            if (l.getString(i).charAt(0) == inicial){
                return i;
            }
        }
        return -1;
    }

    public void remove() {
        throw new UnsupportedOperationException();
    }

    public void set(String s) {
        throw new UnsupportedOperationException();
    }
}
```

Los métodos add(), set() y remove() son opcionales, si no se implementan lanzan esta excepción.

```
import java.util.ListIterator;
import java.util.Iterator;

public class ListaStringArray implements ListaString{
    private String[] lista;
    private int cont;

    public ListaStringArray(int tamaño) {
        lista = new String [tamaño];
        cont  = 0;
    }

    public void addString(String s) {
        lista[cont++] = s;
    }

    public String getString(int index) {
        return lista[index];
    }

    public int size() {
        return cont;
```

Primero se almacena la cadena y luego se incrementa cont.

```
    }

    public ListIterator<String> listIteratorComienzaPor(char c) {
        return new ListIteratorComienzaPor(this, c);
    }

    public ListIterator<String> listIterator() {
        return new ListIteratorArray(this);
    }

    public Iterator<String> iterator(){
        return listIterator();
    }
}
```

Se crea el iterador pasando `this` como argumento del constructor.

`ListIterator` hereda de `Iterator`, por lo que cumple con la especificación.

Comentario: Las excepciones que se lanzan en los iteradores y los valores devueltos en `previousIndex()` y `nextIndex()` cuando no existen más elementos, al igual que las excepciones lanzadas en `set()`, `remove()` y `add()`, vienen especificados en la documentación de `ListIterator<E>` de la API de Java. Para resolver posibles dudas, se recomienda se consulte dicha documentación.

Nota: Fíjese cómo diferentes iteradores permiten realizar distintos recorridos sin añadir los métodos al tipo de datos `ListaString`.

Ejercicio 8.14:

Escriba una aplicación que declare un objeto de tipo ListaString. Cree ese objeto de tipo ListaStringArray con un tamaño máximo de 20 elementos.

- *Añada varias cadenas de texto a la lista.*
- *Haga un recorrido hacia adelante y otro hacia atrás mostrando todos los elementos de la lista. Utilice para ello el iterador devuelto por listIterator().*
- *Haga un recorrido hacia adelante y otro hacia atrás mostrando los elementos de la lista que comiencen por el carácter 'c'. Utilice para ello el iterador devuelto por listIteratorComienzaPor().*
- *Haga un recorrido hacia adelante mostrando todos los elementos de la lista utilizando una versión reducida del for.*

Planteamiento: Se declara el objeto de tipo `ListaString` y se crea de tipo `ListaStringArray`. Como ya se ha visto esto no genera incompatibilidad de tipos, ya que los objetos de tipo `ListaStringArray` son también de tipo `ListaString`.

- Para añadir se utiliza el método `addString()`.
- Para hacer el recorrido por todos los elementos se obtiene un iterador mediante `listIterator()` y mientras que haya siguiente se obtiene y se muestra. Para hacer el recorrido hacia atrás, se hace igual pero comprobando si hay elemento anterior y obteniéndolo.
- Para hacer el recorrido por las cadenas que comiencen por 'c' se obtiene un iterador con el método `listIteratorComienzaPor()` y el recorrido se realiza exactamente igual que el anterior. El iterador se encarga de seleccionar qué cadenas cumplen la condición y cuáles no.
- Para realizar el recorrido con el `for`, no existe problema para utilizar la versión reducida, ya que `ListaString` es también `Iterable`.

Solución:

```
import java.util.ListIterator;
public class Main {
```

```
    public static void main(String[] args) {
        ListaString l;
        l = new ListaStringArray(20); ◄──── ListaStringArray es también ListaString.
        l.addString("hola");
        l.addString("adios");
        l.addString("uno");
        l.addString("dos");
        l.addString("tres");
        l.addString("cuatro");
        l.addString("cinco");
        l.addString("seis");
        l.addString("siete");
        l.addString("ocho");
        l.addString("coche");
        l.addString("camión");

        // recorrido hacia delante y hacia atrás por todos los elementos
        System.out.println("\nListado de todos los elementos ida y vuelta");
        ListIterator<String> it = l.listIterator(); ◄──── Se obtiene el iterador.
       while (it.hasNext()){
            System.out.println(it.next());
       }
       while(it.hasPrevious()){
            System.out.println(it.previous());
       }

        // recorrido hacia delante y hacia atrás los elementos que comienzan por "c"
        System.out.println("\nListado de los elementos que comienzan por c ida y vuelta");
        it = l.listIteratorComienzaPor('c'); ◄──── Se obtiene un iterador diferente,
        while (it.hasNext()){                       pero el recorrido se hace igual.
            System.out.println(it.next());
       }
        while(it.hasPrevious()){
            System.out.println(it.previous());
       }

        // recorrido hacia delante con for
        System.out.println("\nListado de todos los elementos ida");
        for (String s: l){ ◄──── Recorrido con la versión reducida del for.
            System.out.println(s);
       }

    }
}
```

Nota: Cuando se hace el recorrido hacia adelante y hacia atrás, el último elemento sólo se muestra una vez. El recorrido se para en el último elemento y luego muestra el anterior a éste.

Ejercicio 8.15:

Modifique la interfaz ListaString del Ejercicio 8.12 para que, además de heredar de Iterable<String>, herede de Comparable.

Planteamiento: Una interfaz puede heredar de todas las interfaces que se desee. Eso sí, nunca puede heredar de una clase, ya que no puede tener nada implementado. Únicamente hay que modificar la cabecera. `Comparable` también es genérico, por lo que se le pasará como parámetro actual de tipo `ListaString`, ya que una `ListaString` se compara con otra `ListaString`.

```
import java.util.ListIterator;

public interface ListaString extends Iterable<String>, Comparable<ListaString>{
    String getString(int index);
    void addString(String s);
    int size();
    ListIterator<String> listIteratorComienzaPor(char c);
    ListIterator<String> listIterator();
}
```

`ListaString` hereda de `Iterable` y `Comparable`.

Ejercicio 8.16:

Al modificar la interfaz ListaString, la clase ListaStringArray del Ejercicio 8.13 no compila, ya que no implementa el método compareTo() de Comparable. Añadir el método compareTo() de manera que compare dos listas por el número de elementos que contengan.

Planteamiento: El método `compareTo()` recibe un objeto de tipo `ListaString`, y debe devolver 0 si tienen la misma longitud, algún número menor que 0 si la lista que llama al método tiene menos elementos que la que se recibe por parámetro y algún número mayor que 0 si es la lista que llama al método la que tiene más elementos. Por tanto, la implementación consiste en devolver la diferencia entre el número de elementos de las listas.

Solución:

```
public int compareTo(ListaString l){
   return size() - l.size();
}
```

Nota: Este método debe ir dentro de la clase `ListaStringArray`.

Ejercicio 8.17:

Responda a las siguientes preguntas sobre interfaces y clases.

- *¿Puede una interfaz heredar de otra interfaz? ¿Y de varias?*
- *¿Puede una interfaz implementar otra interfaz? ¿Y varias?*
- *¿Puede una clase heredar de una interfaz? ¿Y de varias?*
- *¿Puede una clase implementar una interfaz? ¿Y varias?*
- *¿Puede una clase heredar de otra clase? ¿Y de varias?*
- *¿Puede una clase implementar otra clase? ¿Y varias?*
- *¿Puede una interfaz heredar de una clase? ¿Y de varias?*
- *¿Puede una interfaz implementar una clase? ¿Y varias?*

Planteamiento: Una interfaz nunca implementa nada, luego no implementará ni clases ni interfaces.

Como una interfaz no puede implementar nada, tampoco puede heredar de una clase, ya que entonces heredaría la implementación que tuviese la clase.

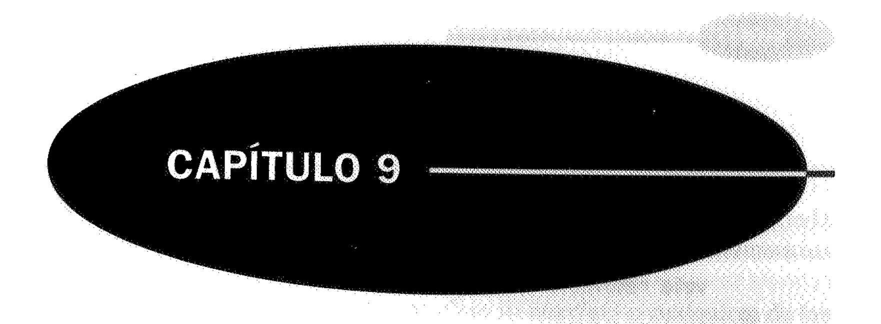
Una clase sólo puede heredar de una clase y puede implementar todas las interfaces que se desee.

De una clase se hereda, una clase no se implementa.

Las clases no heredan de las interfaces, las implementan.

Solución:

• ¿Puede una interfaz heredar de otra interfaz?	*sí*	¿Y de varias?	*sí*
• ¿Puede una interfaz implementar otra interfaz?	*no*	¿Y varias?	*no*
• ¿Puede una clase heredar de una interfaz?	*no*	¿Y de varias?	*no*
• ¿Puede una clase implementar una interfaz?	*sí*	¿Y varias?	*sí*
• ¿Puede una clase heredar de otra clase?	*sí*	¿Y de varias?	*no*
• ¿Puede una clase implementar otra clase?	*no*	¿Y varias?	*no*
• ¿Puede una interfaz heredar de una clase?	*no*	¿Y de varias?	*no*
• ¿Puede una interfaz implementar una clase?	*no*	¿Y varias?	*no*



Genéricos

9.1 GENÉRICOS

En este capítulo se describe un mecanismo para definir clases e interfaces parametrizando su definición de acuerdo con un tipo *genérico*. El tipo concreto se proporcionará cuando se declaren elementos de dicha clase o interfaz. Por tanto, los genéricos son una abstracción en la definición de clases e interfaces.

El ejemplo que encontrará rápidamente y que ya se ha utilizado en el Capítulo 6 es la interfaz `Collection` y sus derivadas. Esta interfaz permite especificar una colección de objetos de un cierto tipo, pero sin especificarlo en la definición, sino en el uso. En efecto, si se imagina una colección de `String` y otra de `Alumno`, es fácil imaginar que el código es igual, pero haciendo referencia a `String` y a `Alumno` según el caso. El siguiente es un ejemplo de uso de la clase `ArrayList` en la que se especifica que se van a guardar objetos de la clase `Alumno`:

```
ArrayList<Alumno> miGrupo = new ArrayList<Alumno>(50);

miGrupo.add(new Alumno("Juan", "García", 1980));
miGrupo.add(new Alumno("Maria", "López", 1981));

Iterator<Alumno> a = miGrupo.iterator();
while (a.hasNext()) {
   Alumno alum = a.next();
   alum.ponGrupo("33", Horario.TARDE);
}
```

En las declaraciones se pone el nombre de la clase genérica y entre ángulos (signos de menor que y de mayor que) el nombre de la clase con la que se parametriza. El resultado es que no se dispone de un objeto `ArrayList` cualquiera, sino un `ArrayList` de alumnos, que se escribe como `ArrayList<Alumno>`. Es decir, `ArrayList` es una clase genérica que toma como parámetro un tipo, en el ejemplo, `Alumno`.

9.2 DEFINICIÓN DE GENÉRICOS

Para ver cómo declarar genéricos, se presenta a continuación la definición del interfaz `Iterator` del paquete `java.util`:

```
public interface Iterator<E> {
    boolean hasNext();

    E next();

    void remove();
}
```

La definición es muy similar a la de cualquier interfaz, excepto por <E>, que quiere decir que este interfaz toma como parámetro un tipo que se denominará E. A este tipo se le llama tipo formal parámetro del genérico. Se puede usar como si existiese ya declarado, con alguna excepción, por ejemplo <E> no puede aparecer en la declaración ni inicialización de un elemento de clase.

9.3 HERENCIA DE GENÉRICOS Y CONVERSIÓN DE TIPOS

Los genéricos se pueden heredar como cualquier otra clase o interfaz. Las reglas de extensión son las mismas que se aplican en la herencia, como se explicó en los Capítulos 5 y 8. No obstante, las reglas de compatibilidad de tipos requieren una explicación detallada. En el siguiente código, se asignan referencias de genéricos:

```
List<Alumno>  ls = new Vector<Alumno>();
List<Persona> lp = ls;
```

La primera línea asigna a una lista de alumnos una instancia de vectores de alumnos. Efectivamente, Vector<E> extiende List<E>, siendo tipos compatibles. El problema aparece en la siguiente línea: el tipo List<Persona> no es un tipo compatible con List<Alumno>. Para entenderlo, baste pensar que en el objeto lp, de tipo List<Persona>, se pueden insertar objetos de la clase Persona, que no son compatibles con los objetos de ls, que al ser List<Alumno>, sólo admite insertar objetos de la clase Alumno.

9.4 COMODINES

El ejemplo de la sección anterior muestra una necesidad en el uso de genéricos. Dado que las instanciaciones de genéricos sólo son compatibles si el tipo parametrizado es el mismo, no habría forma de escribir código que sea válido para un tipo genérico si no se sabe cómo se instanciará. El siguiente es un ejemplo que intenta imprimir los datos de las personas de una lista:

```
void imprime(List<Persona> c) {
    for (Persona p : c) {
        System.out.println(p);
    }
}
```

Sin embargo, el método anterior no es muy útil pues no se puede utilizar para imprimir una lista de alumnos, de profesores, de bedeles, dado que List<Persona> no es una clase compatible con List<Alumno>, List<Profesor>, etc. No obstante, se incluye un mecanismo denominado de "comodines" que permite generalizar el uso de genéricos. El siguiente código permite imprimir los objetos de una lista:

```
void imprimeTodo(List<?> c) {
    for (Object p : c) {
        System.out.println(p);
```

```
        }
    }
```

El comodín se representa con el carácter de interrogación '?'. De esta forma, `List<?>` representa una lista instanciada por cualquier tipo. Sin embargo, si se observa el bucle `for`, los objetos de la lista sólo se pueden usar como referencias a `Object`.

Los comodines pueden ser *ligados*, especificando que los tipos sean de una clase (o interfaz) o cualquiera de las clases derivadas. Esto se escribe como

```
List<? extends nombreClase>
List<? extends nombreInterfaz>
```

En el siguiente ejemplo, se presenta un método que imprime los apellidos y año de nacimiento de una lista de personas.

```
void imprimePersonas(List<? extends Persona> c) {
    for (Persona p: c) {
        System.out.println(p.dameApellidos() +
                           " " + p.dameAñoNacimiento());
    }
}
```

9.5 MÉTODOS GENÉRICOS

Sólo falta ver cómo declarar métodos que usen la clase o el interfaz genérico. Es posible especificando que un método va a usar un tipo genérico, como presenta el siguiente ejemplo:

```
public static <T> void inserta(List<T> lp, T o) {
    lp.add(o);
}
```

Se define un método genérico, `inserta`. Se indica que es genérico añadiendo la especificación de parámetros de tipo (uno o más) antes del valor de retorno del método, en este caso, `<T>`. El método acepta una lista genérica, `List<T>` y un objeto de tipo `T`. De hecho, se puede usar el tipo `T` en el cuerpo del método, como muestra el siguiente ejemplo que inserta un array de alumnos en una lista:

```
public static <T> void inserta(List<T> lp, T[] a) {
    for (T o : a) // recorre el array a
        lp.add(o);
}
```

Así como con `? extends T` se pone un límite superior en la jerarquía de tipos aceptada por un método genérico, o en una declaración de clase o interfaz genérico, también es necesario poder poner un límite inferior en los tipos aceptados. Para ello se utiliza la sintaxis:

```
TreeSet(Comparator<? super E> c);
```

En este ejemplo se indica que se acepta que la clase para la ordenación de los elementos sea un comparador del tipo `E` o de cualquiera de sus ancestros. Por último, indicar que es posible mezclar el uso de métodos genéricos y comodines ligados.

La interfaz Collections muestra un buen ejemplo de ello, como el método copy:

```
public static <T> void copy(List<? super T> dest, List<? extends T> src)
```

Este método acepta dos listas de elementos y copia todos los elementos de la lista src en dest. Para ello, el método se declara genérico, parametrizado por el tipo T. La lista origen se restringe a T y sus derivados. La lista destino puede ser T o cualquiera de sus ancestros. De esta forma, se obtiene toda la flexibilidad en la conversión ascendente de tipos.



Problemas resueltos

Genéricos

Ejercicio 9.1:

Escriba una clase Pila genérica usando para ello un atributo de tipo LinkedList. La clase Pila tendrá los siguientes métodos:

- *estáVacía(): devuelve true si la pila está vacía y false en caso contrario.*
- *extraer(): devuelve y elimina el primer elemento de la colección.*
- *primero(): devuelva el primer elemento de la colección.*
- *añadir(): añade un objeto por el extremo que corresponda.*
- *toString(): devuelve en forma de String la información de la colección.*

Planteamiento: Aprovechando que la clase `LinkedList` es genérica, sólo se tiene que pasar como tipo genérico del atributo `LinkedList` como parámetro genérico de la clase `Pila<E>`.

- `estáVacía()`: utiliza el método `isEmpty()` de `LinkedList`.
- `extraer()`: utiliza el método `removeFirst()`, devuelve lo mismo que este método
- `primero()`: utiliza el método `getFirst()`, devuelve lo mismo que es método
- `añadir()`: utiliza el método `addFirst()`.
- `toString()`: devuelve lo mismo que el método `toString()` de `LinkedList`.

Solución:

```
import java.util.LinkedList;

public class Pila <E>{   // Tipo genérico de la Pila que se pasa al LinkedList.
    private LinkedList<E> lista;

    public Pila() {
        lista = new LinkedList<E>();
    }

    public void añadir(E o){   // Recibe un objeto de tipo genérico E.
        lista.addFirst(o);
    }

    public E primero(){   // Devuelve un objeto de tipo genérico E.
        return lista.getFirst();
    }

    public E extraer(){
        return lista.removeFirst();
    }

    public boolean estáVacía(){
        return lista.isEmpty();
    }
```

```
    public String toString(){
        return lista.toString();
    }
}
```

Comentario: Se podría hacer exactamente igual añadiendo por el final y extrayendo por el final. El método `toString()` en este caso devolvería los elementos comenzando por el último de la pila, aunque se podría implementar el método `toString()` de nuevo.

Ejercicio 9.2:

Implemente una pila utilizando como atributos un array genérico y un entero que cuente el número de objetos insertados. La clase se debe llamar PilaArray y tiene los mismos métodos que la pila del ejercicio anterior.

Planteamiento: La clase `PilaArray` tiene un parámetro genérico `E`.

Esta clase tiene como atributos un `array` del mismo tipo genérico `E` que la clase y un entero que sirve de contador de objetos. El constructor recibe por parámetro el tamaño máximo de la pila. El método `estáVacía()` comprueba si el contador es 0.

El método `añadir()` recibe por parámetro un objeto de tipo `E`, comprueba que hay espacio libre y, si es así, lo añade en la celda que indica el contador. Posteriormente incrementa el valor del contador. Si se ha añadido devuelve `true`. Si no se ha podido añadir devuelve `false`.

`primero()`: si está vacía lanza `NoSuchElementException`. Si no está vacía devuelve el elemento que está en la celda indicada por el contador - 1. El método se declara de tipo `E` ya que los objetos que va a devolver son del tipo parámetro.

`extraer()`: si está vacía lanza `NoSuchElementException`. Si no está vacía decrementa el contador y devuelve el elemento que está en la celda indicada por el contador después de decrementarse. Extraer se declara también de tipo `E`.

`NoSuchElementException`: hereda de `RuntimeException`, por lo que no que se declara en la cláusula `throws` del método.

Se sobreescribe el método `toString()` de manera que devuelva los objetos encerrados entre corchetes y separados por comas, comenzando por el último elemento insertado y terminando por el primero.

Solución:

```
import java.util.NoSuchElementException;
public class PilaArray <E>{

    private E[] array;
    private int contador;

    public PilaArray(int tamañoMáximo) {
        array = (E[]) new Object[tamañoMáximo];
        contador = 0;
    }

    public boolean estáVacía(){
        return contador == 0;
    }

    public boolean añadir(E o){
        if (contador == array.length)
            return false;
        else{
            array[contador] = o;
```

Tipo genérico de la `Pila` que se pasa al `LinkedList`.

Array de tipo `E` genérico.

No se permite crear arrays de un tipo genérico.

Conversión de tipo a `E[]`.

```
                contador++;
                return true;
            }
        }

        public E primero(){
            if (estáVacía())
                throw new NoSuchElementException();
            else
                return array[contador-1];
        }

        public E extraer(){
            if (estáVacía())
                throw new NoSuchElementException();
            else
                return array[--contador];
        }

        public String toString(){
            String s = "[";
            for (int i = contador - 1; i >= 0; i--){
                s += array[i].toString()+",";
            }
            if (!estáVacía())
                return s.substring(0,s.length()-1)+"]";
            else
                return s + "]";
        }
}
```

Para eliminar la última coma.

Nota: No se pueden crear `arrays` de un tipo genérico, aunque sí se pueden declarar. Para superar esta dificultad, se crea un `array` de `Object` y se transforma a array del tipo genérico `E`. Esta conversión de tipo muestra un `warning` en tiempo de compilación que no supone ningún problema, ya que sólo se añaden elementos al `array` mediante el método `añadir()` que recibe un objeto del tipo genérico `E`.

Ejercicio 9.3:

Escriba una aplicación que:

a) Cree una PilaArray de String con tamaño 10.
b) Introduzca tres elementos en la pila.
c) Muestre a pantalla el contenido de la pila.
d) Muestre el primer carácter de la primera cadena de la pila.
e) Cree una PilaArray de Integer con tamaño 10.
f) Introduzca tres elementos en la pila.
g) Muestre a pantalla el contenido de la pila.
h) Calcule la suma de los elementos que hay en la pila, extrayéndolos de ella.

Planteamiento:

a) Para crear una `PilaArray` de un tipo determinado, el tipo se pasa entre < >. Tanto en la declaración como en la creación.

b) Posteriormente se introducen tres objetos de tipo `String`, en este caso no hay conversión a la superclase (`Object`) ya que la pila es de tipo `String`.
c) Para mostrar el contenido en la pantalla se utiliza el método `toString()`, aunque no es necesario llamarlo explícitamente.
d) Para obtener el primer elemento de la pila, se utiliza el método `primero()` que devuelve un `String`. De esta cadena de texto se obtiene el primer carácter utilizando el método `charAt()`.
e) Para crear la `PilaArray` de `Integer`, se pasa `Integer` entre los < >, tanto en la declaración como en la creación.
f) Se puede pasar como argumento en `añadir()` un `int`, ya que se hace una conversión automática a `Integer`.
g) Para mostrar el contenido a pantalla se utiliza el método `toString()`.
h) Para sumar todos mientras se extraen, se realiza un bucle que se repite mientras la pila no está vacía. No es necesario llamar a los métodos de la clase `Integer` recibida, ya que se convierte automáticamente a `int`.

Solución:

```
public class Main {

    public static void main(String[] args) {
        PilaArray<String> p1 = new PilaArray<String>(10);   ← Se pasa como parámetro genérico el tipo deseado.
        p1.añadir("uno");
        p1.añadir("dos");
        p1.añadir("tres");
        System.out.println(p1.toString());
        System.out.println(p1.primero().charAt(0));   ← Se llama a charAt() con el objeto devuelto, que es de tipo String.

        PilaArray<Integer> p2 = new PilaArray<Integer>(10);
        p2.añadir(1);   ← Se convierten automáticamente a Integer.
        p2.añadir(2);
        p2.añadir(3);
        System.out.println(p2);
        int suma = 0;
        while(!p2.estáVacía()){
            suma += p2.extraer();   ← Se convierten automáticamente a int.
        }
        System.out.println("La suma es " + suma);
    }
}
```

Nota: El tipo genérico nunca puede ser un tipo primitivo.

Pruebe a introducir en `p1` un `Integer` y en `p2` un `String`. Obtendrá un error de compilación. Esta es una de las diferencias con respecto a tener una `Pila` de `Object`, que aceptaría siempre cualquier objeto.

La otra diferencia importante es que los métodos `primero()` y `extraer()` devuelven objetos del tipo que se ha especificado al crear la pila, por tanto no es necesaria la conversión de tipo.

Ejercicio 9.4:

Escriba una clase Matriz genérica con los siguientes métodos:

- *constructor que recibe por parámetro el número de filas y columnas de la matriz.*
- *set() recibe por parámetro la fila, columna y el elemento a insertar. El elemento es de tipo genérico. Este método inserta el elemento en la posición indicada.*
- *get() recibe por parámetro la fila y la columna. Devuelve el elemento en esa posición. El elemento devuelto es del tipo genérico.*

- *columnas () devuelve el número de columnas de la matriz.*
- *filas () devuelve el número de filas de la matriz.*
- *toString() devuelve en forma de String la información de la matriz.*

Planteamiento: La matriz tiene un parámetro genérico `E`. El método `set()` recibe por parámetro además de la posición un elemento de tipo `E`. Y el método `get()` devuelve un elemento de tipo `E`.

El atributo donde se guarda la información es un array bidimensional de tipo `E`. Al crearlo, como Java no permite crear arrays de un tipo genérico, se crea un array bidimensional de `Object` y luego se convierte a array bidimensional de `E`.

El método `toString()` utiliza un `String` auxiliar donde se acumula la información de cada celda, para separar unas celdas de otras se puede utilizar el carácter `'\t'`, o también un espacio.

Solución:

```
public class Matriz<E> {

    private E[][] tabla;

    public Matriz(int filas, int columnas) {
        tabla = (E[][]) new Object[filas][columnas];
    }

    public void set(int fila, int columna, E elemento){
        tabla[fila][columna] = elemento;
    }

    public E get(int fila, int columna){
        return tabla[fila][columna];
    }

    public int columnas(){
        return tabla[0].length;
    }

    public int filas(){
        return tabla.length;
    }

    public String toString(){
        String s = "";
        for (int i = 0; i < tabla.length; i++){
            for (int j = 0; j < tabla[0].length; j++){
                s += tabla[i][j] + "\t";
            }
            s += "\n";
        }
        return s;
    }
}
```

Parámetro genérico E.

No se pueden crear arrays de un tipo genérico.

Conversión de tipo a `E[][]`.

`String` auxiliar para almacenar la información.

Nota: Las celdas de la matriz no se inicializan a ningún valor, por lo que el resultado de `get()` podría ser `null`. Al llamar al método `toString()` también cabe la posibilidad de que devuelva `null` para alguna de las celdas.

Ejercicio 9.5:

Escriba una aplicación que:

a) Cree una matriz de Integer de 4 filas y 2 columnas.
b) Rellénela con números consecutivos comenzando por el número 1.
c) Muestre por pantalla la matriz.
d) Muestre por pantalla el contenido de la celda en la fila 0 columna 1.

Planteamiento: Se crea la matriz de `Integer` pasando las dimensiones por parámetro.

Para rellenarla, se realizan dos `for` anidados que vayan pasando por todas las celdas de la matriz. Para el número a insertar se declara una variable de tipo `int` que se incrementa después de cada inserción.

Para mostrar la matriz se utiliza el método `toString()`, aunque no es necesario llamarlo explícitamente.

Para acceder a una celda determinada se utiliza el método `get()`.

Solución:

```
public class Main {

    public static void main(String[] args) {
        Matriz<Integer> m = new Matriz<Integer>(4,2);
        int num = 1;

        for (int i = 0; i < m.filas(); i++){
            for (int j = 0; j < m.columnas(); j++){
                m.set(i,j,num++);
            }
        }
        System.out.println(m);
        System.out.println(m.get(0, 1));

    }
}
```

Se pasa como parámetro genérico el tipo deseado.

Ejercicio 9.6:

Escriba una interfaz ColeccionSimple genérica con los siguientes métodos:

- *estáVacía(): devuelve true si la pila está vacía y false en caso contrario.*
- *extraer(): devuelve y elimina el primer elemento de la colección.*
- *primero(): devuelva el primer elemento de la colección.*
- *añadir(): añade un objeto por el extremo que corresponda.*

Planteamiento: Se realiza igual que la interfaz `ColeccionSimple` realizada en el Ejercicio 8.1 del Capítulo 8 sobre interfaces, pero declarando el tipo genérico en la cabecera de la interfaz. Los parámetros y valores de retorno de los métodos son del tipo genérico en lugar de ser `Object`.

Solución:

```
public interface ColeccionSimple <E>{
    boolean estáVacía();
    boolean añadir(E o);
    E primero();
    E extraer();
}
```

Comentario: La clase PilaArray<E> sería una implementación de esta interfaz. Para hacer que PilaArray<E> implemente ColeccionSimple<E> sólo habría que cambiar la cabecera de la clase por la siguiente:

```
public class PilaArray<E> implements ColeccionSimple<E>
```

El resto de la clase sería igual.

Ejercicio 9.7:

Escriba una clase genérica ListaOrdenada con un tipo parametrizado E que sea Comparable. La clase debe tener los siguientes métodos:

- *Un constructor*
- *void add(E o)*
- *E get(int index)*
- *int size()*
- *boolean isEmpty()*
- *boolean remove(E o)*
- *int indexOf(E o)*
- *String toString()*

Planteamiento: Para implementar esta ListaOrdenada, se declara un atributo de tipo List<E>.

Todos los métodos excepto el método add() consisten en una llamada al método correspondiente de List.

En el método add() se recorre el atributo comparando el objeto a insertar con los objetos de la lista. Como ya se conoce que la clase de los objetos que se insertan implementa Comparable, se puede usar el método compareTo() para comparar los objetos. Cuando se encuentra su posición se llama al método add(int index, E o) de List, para añadirlo en la posición específica. Este método ya se encarga de desplazar a la derecha el resto de los elementos de la lista. Si se termina de recorrer la lista sin encontrar su posición es porque hay que añadirlo al final, para esto se llama al método add(E o) de List.

Solución:

```
import java.util.ArrayList;
import java.util.List;
public class ListaOrdenada <E extends Comparable<E>>{ <---
    private List<E> lista; <---

    public ListaOrdenada() {
        lista = new ArrayList<E>();
    }

    public boolean add(E obj){
        for (int i = 0; i < lista.size(); i++){
            if (obj.compareTo(lista.get(i))<0){ <---
                lista.add(i, obj);
                return true;
            }
        }
        lista.add(obj);
        return true;
    }

    public E get(int index){
```

Se restringe el tipo genérico a las clases que implementan Comparable.

El parámetro se declara List del mismo tipo genérico E.

Los objetos son Comparables, por eso se puede llamar a compareTo().

```
        return lista.get(index);
    }

    public int size(){
        return lista.size();
    }

    public boolean remove(E obj){
        return lista.remove(obj);
    }

    public boolean isEmpty(){
        return lista.isEmpty();
    }

    public int indexOf(E obj){
        return lista.indexOf(obj);
    }

    public String toString(){
        String s="";
        for (int i=0; i<lista.size(); i++){
            s+=lista.get(i)+"\n";
        }
        return s;
    }
}
```

El resto de los métodos delegan en los del atributo `lista`.

Comentario: El método `indexOf()` de las listas busca un elemento en la lista. Para comparar el objeto que entra por parámetro con los objetos que están en la lista utiliza el método `equals()`. Si el usuario no quiere que se utilice el método `equals()` de la clase `Object`, debe sobreescribirlo.

Recuerde que el método `equals()` de `Object` sólo compara referencias a objetos, como el operador ==.

Ejercicio 9.8:

Escriba una clase ArrayListOrdenado que herede de ArrayList. Esta clase sólo debe aceptar, como parámetro genérico de tipo, clases que implementen Comparable. La clase ArrayListOrdenado debe sobreescribir el método add(E obj) para que añada los elementos en orden.

Planteamiento: La clase `ArrayListOrdenado` hereda de `ArrayList`. Recibe como parámetro genérico un tipo `E` que implemente `Comparable`. Este parámetro `E` se pasa en la cabecera de la clase a `ArrayList`.

En el método `add()` se recorre el atributo comparando el objeto a insertar con los objetos de la lista. Como ya se conoce que la clase de los objetos que se insertan implementa `Comparable`, se puede usar el método `compareTo()` para comparar los objetos. Cuando se encuentra su posición se llama al método `add(int index, E o)` heredado de `ArrayList`, para añadirlo en la posición específica. Este método ya se encarga de desplazar a la derecha el resto de los elementos de la lista. Si se termina de recorrer la lista sin encontrar su posición es porque hay que añadirlo al final, para esto se llama al método `add(E o)` de la superclase.

Solución:

La misma `E` que para `ArrayListOrdenado`, por tanto implementan `Comparable`.

Se restringe el tipo genérico a las clases que implementan `Comparable`.

```
import java.util.ArrayList;
public class ArrayListOrdenado <E extends Comparable<E>> extends ArrayList<E>{
```

```
        public ArrayListOrdenado() {
        }

        public boolean add(E obj){
            for (int i=0; i<size(); i++){
                if (obj.compareTo(get(i))<0){
                    add(i, obj);
                    return true;
                }
            }
            super.add(obj);
            return true;
        }
}
```

Los objetos son Comparables, por eso se puede llamar a compareTo().

Llamada a add() de la superclase.

Comentario: Aunque se haya sobreescrito este método add(), la lista podría no mantenerse ordenada si se utilizasen otros métodos para añadir heredados de ArrayList. Para solucionarlo se podrían sobreescribir todos los métodos que permitan añadir y modificar. Otra solución posiblemente mejor es utilizar una relación de composición como en la clase ListaOrdenada.

Se recomienda usar relaciones de composición si no se quieren heredar todos los métodos.

Ejercicio 9.9:

Escriba una aplicación que cree una ListaOrdenada que almacene objetos de la clase Proyecto que se realizó en el Ejercicio 8.10 del Capítulo 8 sobre interfaces. Añada tres proyectos y muestre la lista por pantalla.

Planteamiento: Se declara y crea la ListaOrdenada pasando como parámetro genérico el tipo Proyecto. Proyecto, al implementar Comparable, se acepta como tipo de dato de la ListaOrdenada.

Cuando se añadan elementos, se ordenarán por el código del proyecto, ya que fue así como se implementó el método compareTo() en Proyecto.

Para añadir proyectos, se utiliza el método add(), y para mostrarlos, el método toString(), aunque no es necesario llamarlo explícitamente.

Solución:

```
public class Main {

    public static void main(String[] args) {
      ListaOrdenada<Proyecto> proyectos = new ListaOrdenada<Proyecto>();
      proyectos.add(new Proyecto("SIS-04039","Editor de partituras musicales", 2005, Carrera.SIS));
      proyectos.add(new Proyecto("INF-04003","Reconocimiento de voz", 2004, Carrera.INF));
      proyectos.add(new Proyecto("SIS-03014","Herramientas SAP", 2004, Carrera.SIS));
      System.out.println(proyectos);
    }
}
```

Proyecto implementa Comparable.

Comentario: Si se intenta instanciar un objeto de tipo ListaOrdenada utilizando como tipo una clase que no implemente Comparable, se producirá un error en tiempo de compilación.

Ejercicio 9.10:

Escriba una interfaz genérica Operable que sea genérica y que declare las cuatro operaciones básicas: suma, resta, producto y división.

Planteamiento: Se escribe la interfaz `Operable` con un tipo genérico `E`. Todos los métodos de esta interfaz reciben un objeto de tipo `E` y devuelven un objeto de tipo `E`. De esta manera se opera el objeto que llame al método con el que se recibe por parámetro y se devuelve el resultado. Ambos operandos y el resultado son del mismo tipo.

Solución:

```
public interface Operable <E>{
    E suma(E obj);   <-- Todas las operaciones reciben y devuelven un objeto de tipo genérico.
    E resta(E obj);
    E producto(E obj);
    E división(E obj);
}
```

Ejercicio 9.11:

Realice una clase de nombre Racional que implemente la interfaz Operable.

Además de las cuatro operaciones, debe sobreescribir el método toString(). No se deben permitir Racionales con denominador 0.

Planteamiento: La clase `Racional` tiene como atributos `numerador` y `denominador` de tipo entero. Las operaciones se realizan entre racionales, por tanto implementa `Operable<Racional>`. El método `toString()` consiste en devolver el `numerador`, una barra de dividir y el `denominador`.

Para evitar que existan racionales con denominador 0, se controla en el constructor y en la división, ya que en las demás operaciones el denominador siempre es el producto de los denominadores.

En el constructor se lanza una excepción de tipo `IllegalArgumentException`, y en la división de tipo `ArithmeticException`.

Solución:

```
public class Racional implements Operable<Racional>{   <-- Racional implementa Operable<Racional>.
    private int num;
    private int den;

    public Racional(int numerador, int denominador){
        if (denominador == 0){   <-- El denominador no puede ser 0.
            throw new IllegalArgumentException();
        }
        num = numerador;
        den = denominador;
    }

    public int getNumerador(){
        return num;
    }

    public int getDenominador(){
        return den;
    }

    public Racional suma (Racional r){   <-- Implementación de los métodos de la interface Operable.
        int numRes = num * r.getDenominador() + r.getNumerador() * den;
        int denRes = den * r.getDenominador();
        return new Racional(numRes, denRes);
```

```
    }

    public Racional resta (Racional r){
        int numRes = num * r.getDenominador() - r.getNumerador() * den;
        int denRes = den * r.getDenominador();
        return new Racional(numRes, denRes);
    }

    public Racional producto (Racional r){
        int numRes = num * r.getNumerador();
        int denRes = den * r.getDenominador();
        return new Racional(numRes, denRes);
    }

    public Racional división (Racional r){
        int numRes = num * r.getDenominador();
        int denRes = den * r.getNumerador();
        if (denRes == 0){
            throw new ArithmeticException();
        }
        return new Racional(numRes, denRes);
    }

    public String toString(){
        return num + "/" + den;
    }
}
```

Por simplicidad no se simplifica el resultado de ninguna operación.

Si el resultado de la operación es un racional con denominador 0 se lanza una excepción.

Comentario: Las operaciones se han implementado de la manera más sencilla. El resultado está sin simplificar. Si se considera necesario se podría escribir un método `simplificar()` y llamarlo después de cada operación.

Ejercicio 9.12:

Escriba un método que reciba List de objetos de tipo Poligono y calcule la suma de sus perímetros. La clase Poligono que se utilizará será la del Ejercicio 5.5 del Capítulo 5 sobre extensión de clases.

Plantemiento: El método recibe por parámetro una `List` genérica de tipo `Poligono`. Dentro del método se declara una variable suma que se inicializa a 0. Se realiza un bucle que recorre todos los polígonos de la lista y va acumulando el resultado del perímetro a la variable suma. Por último se devuelve el resultado.

Solución:

```
public static double sumaPerímetros(List<Poligono> lista){
    double suma = 0;
    for(Poligono p: lista){
        suma += p.perimetro();
    }
    return suma;
}
```

Para recorrer todos los objetos de la lista, que van a ser `Poligonos`.

Comentario: Los métodos de estos ejercicios se declaran `static` porque se suponen en la misma clase que el método `main()`, y así se pueden invocar directamente desde el `main()` sin crear ningún objeto.

Los métodos funcionan igual si se elimina la cláusula `static` y se utiliza un objeto para invocarlos.

Ejercicio 9.13:

Escriba una aplicación en la que:

- *Se creen 3 Poligonos.*
- *Se cree un ArrayList de Poligono.*
- *Se añadan esos tres polígonos.*
- *Se llame al método sumaPerímetro() y se imprima el resultado por pantalla.*

Suponer que el método sumaPerímetro() está en la misma clase que el main()

Planteamiento: Lo primero que se crean son los tres arrays de `Punto`. Éstos se pasan como argumentos en los constructores de los polígonos.

Una vez creados los tres polígonos, se crea el `ArrayList` de `Poligono`. Posteriormente se añaden los tres `Poligonos` con el método `add()`.

Por último, se muestra por pantalla el resultado de llamar al método `sumaPerímetro()`. El método al ser `static` y estar en la misma clase, se puede invocar directamente desde el `main()`, sin crear ningún objeto.

Solución:

```
public static void main(String args[]){
    Punto[] vertices1 = {new Punto(0,0), new Punto(0,1), new Punto(1,1), new Punto(1,0)};
    Punto[] vertices2 = {new Punto(0,0), new Punto(0,3), new Punto(4,0)};
    Punto[] vertices3 = {new Punto(1,1), new Punto(1,4), new Punto(5,4), new Punto(6,1)};
    Poligono p1 = new Poligono(vertices1);
    Poligono p2 = new Poligono(vertices2);
    Poligono p3 = new Poligono(vertices3);
    ArrayList<Poligono> listaPoligonos = new ArrayList<Poligono>();
    listaPoligonos.add(p1);
    listaPoligonos.add(p2);
    listaPoligonos.add(p3);

    System.out.println(sumaPerímetros(listaPoligonos)); ◄——  Recibe una lista de polígonos
                                                             y devuelve un double.
}
```

Ejercicio 9.14:

El método sumaPerímetros() está escrito para recibir List de Poligono. Aunque la clase PoligonoColoreado del Ejercicio 5.7 del Capítulo 5 hereda de Poligono, el método no está preparado para recibir List de PoligonoColoreado. List<PoligonoColoreado> no hereda de List<Poligono>, y por tanto un intento de llamar a este método con la lista de PoligonoColoreado daría un error de compilación.

El ejercicio consiste en modificar el método sumaPerímetros() para que acepte List de Poligono y cualquier clase que herede de Poligono.

Planteamiento: El parámetro del método se modifica por el siguiente: `List<? extends Poligono>`. Usando un comodín de esta manera aceptará tanto listas de `Poligono`, ya que se entiende que los objetos de tipo `Poligono` cumple con el comodín, como de cualquier clase que herede de ésta.

El resto del método es igual. El objeto para recorrer la lista se puede declarar de tipo `Poligono` ya que todos los elementos de la lista heredan de `Poligono`.

Acepta listas que almacenen cualquier objeto `Poligono` o que herede de `Poligono`.

Solución:

```
public static double sumaPerímetros(List<? extends Poligono> lista){
    double suma = 0;
    for(Poligono p: lista){
```

```
            suma += p.perimetro();
        }
        return suma;
    }
```

Comentario: El método sumaPerímetros() tal como estaba escrito antes, podría haber recibido una List de Poligono rellena de objetos de tipo PoligonoColoreado, pero no es lo que se pide en el ejercicio.

Ejercicio 9.15:

Modifique el método main() del Ejercicio 9.13 para que, además, cree un ArrayList con tres objetos de tipo PoligonoColoreado y llame al método sumaPerímetros() del ejercicio anterior.

Planteamiento: Tras el código del Ejercicio 9.13, se utilizan los arrays de Punto ya existentes para crear los objetos de tipo PoligonoColoreado.

Una vez creados los tres objetos PoligonoColoreado, se crea el ArrayList de PoligonoColoreado. Posteriormente se añaden los tres objetos de tipo PoligonoColoreado con el método add().

Por último, se muestra por pantalla el resultado de llamar al método sumaPerímetro(). El método al ser static y estar en la misma clase, se puede invocar directamente desde el main() sin crear ningún objeto.

Solución:

```
public static void main(String args[]){
    Punto[] vertices1 = {new Punto(0,0), new Punto(0,1), new Punto(1,1), new Punto(1,0)};
    Punto[] vertices2 = {new Punto(0,0), new Punto(0,3), new Punto(4,0)};
    Punto[] vertices3 = {new Punto(1,1), new Punto(1,4), new Punto(5,4), new Punto(6,1)};
    Poligono p1 = new Poligono(vertices1);
    Poligono p2 = new Poligono(vertices2);
    Poligono p3 = new Poligono(vertices3);
    ArrayList<Poligono> listaPoligonos = new ArrayList<Poligono>();
    listaPoligonos.add(p1);
    listaPoligonos.add(p2);
    listaPoligonos.add(p3);

    System.out.println(sumaPerímetros(listaPoligonos));
    /*Poligono Coloreado*/
    PoligonoColoreado pc1 = new PoligonoColoreado(vertices1,Color.black);
    PoligonoColoreado pc2 = new PoligonoColoreado(vertices2,Color.blue);
    PoligonoColoreado pc3 = new PoligonoColoreado(vertices3,Color.green);
    ArrayList<PoligonoColoreado> listaPoligonosCol;
    listaPoligonosCol = new ArrayList<PoligonoColoreado>();   <-- ArrayList de objetos PoligonoColoreado.
    listaPoligonosCol.add(pc1);
    listaPoligonosCol.add(pc2);
    listaPoligonosCol.add(pc3);
    System.out.println(sumaPerímetros(listaPoligonosCol));   <-- ArrayList implementa List y PoligonoColoreado hereda de Poligono.
}
```

Ejercicio 9.16:

Escriba una clase ComparadorPoligono que implemente Comparator<Poligono> y compare dos Poligonos por su perímetro.

Planteamiento: La interfaz Comparator es genérica y recibe el tipo de objetos que va a comparar. Esta interfaz tiene el método compare() que recibe los dos objetos.

El método perímetro() devuelve un double, así que no se puede devolver directamente la diferencia entre los dos perímetros, por lo que se realiza una comparación utilizando una sentencia if-else if.

Solucion:

```
import java.util.*;
public class ComparadorPoligonos implements Comparator<Poligono>{
    public int compare(Poligono p1, Poligono p2){
        if(p1.perimetro() < p2.perimetro()){
            return -1;
        }else if (p1.perimetro() > p2.perimetro()){
            return 1;
        }else{
            return 0;
        }
    }
}
```

Tipo de objetos que se van a comparar.

Ejercicio 9.17:

Escriba un método que reciba por parámetro una List de cualquier tipo y un comparador de cualquier superclase del tipo de la lista. Debe devolver el mayor elemento de la lista según el comparador.

Planteamiento: El método es genérico. Se declara un tipo genérico T. El objeto a devolver es de este tipo T, la lista es de tipo T y el Comparator es del tipo <? super T>.

En la implementación se declara un objeto de tipo T para almacenar el mayor y se inicializa al primer elemento de la lista. Recorre la lista comparando los elementos con el comparador y si encuentra alguno mayor lo actualiza. Por último devuelve el objeto mayor.

Solución:

```
public static <T> T mayor(List<T> lista, Comparator<? super T> comparador){
    T mayor = lista.get(0);
    for(T elemento: lista){
        if(comparador.compare(mayor, elemento)<0){
            mayor = elemento;
        }
    }
    return mayor;
}
```

Puede recibir cualquier comparador de la superclase de los elementos de la lista.

Aviso: Habría que comprobar que la lista recibida no es una referencia a null y tiene al menos un objeto para poder obtener el primero con lista.get(0). En caso contrario se lanzaría una excepción. Se ha omitido por no complicar el método más de lo necesario.

Ejercicio 9.18:

Escriba una aplicación en la que se cree una lista de PoligonosColoreados, se introduzcan tres Poligonos Coloreados y se obtenga el mayor de ellos teniendo en cuenta su perímetro. Suponga que el método mayor() está en la misma clase que el main().

Planteamiento: En la aplicación se deben crear varios objetos de tipo PoligonoColoreado, crear un ArrayList de tipo PoligonoColoreado y añadir los objetos a la lista.

Posteriormente se crea un objeto de tipo ComparadorPoligonos del Ejercicio 9.16. Por último, se llama al método mayor() pasando como argumento la lista y el comparador.

Solución:

```
public static void main(String args[]){
    Punto[] vertices1 = {new Punto(0,0), new Punto(0,1), new Punto(1,1), new Punto(1,0)};
    Punto[] vertices2 = {new Punto(0,0), new Punto(0,3), new Punto(4,0)};
    Punto[] vertices3 = {new Punto(1,1), new Punto(1,4), new Punto(5,4), new Punto(6,1)};

    PoligonoColoreado pc1 = new PoligonoColoreado(vertices1,Color.black);
    PoligonoColoreado pc2 = new PoligonoColoreado(vertices2,Color.blue);
    PoligonoColoreado pc3 = new PoligonoColoreado(vertices3,Color.green);
    ArrayList<PoligonoColoreado> listaPoligonosCol;
    listaPoligonosCol = new ArrayList<PoligonoColoreado>();
    listaPoligonosCol.add(pc1);
    listaPoligonosCol.add(pc2);
    listaPoligonosCol.add(pc3);
    ComparadorPoligonos comparador = new ComparadorPoligonos();
    PoligonoColoreado mayor = mayor(listaPoligonosCol, comparador);
    System.out.println("el mayor es " + mayor);
}
```

comparador es un objeto que implementa Comparator<Poligono>. Poligono es superclase de PoligonoColoreado.

Ejercicio 9.19:

Escriba un método que sume todos los elementos de una lista genérica de objetos que implementen Operable y devuelva el resultado.

Planteamiento: La lista puede ser de cualquier tipo que implemente Operable, es decir puede que no sea una lista de Operable sino de Racionales.

Además el método devuelve un objeto del mismo tipo que los elementos de la lista. Por tanto hay que escribir un método genérico, que tiene como parámetro T extends Operable<T>.

El método se implementa con un acumulador que se inicializa al valor de la primera celda, el resto se ván acumulando en un for que recorre la lista (menos el primer elemento) y finalmente se devuelve el acumulador.

Solución:

```
public static <T extends Operable<T>> T sumaTodos(List<T> lista){
    T resultado = lista.get(0);
    for (int i=1; i<lista.size(); i++){
        resultado = resultado.suma(lista.get(i));
    }
    return resultado;
}
```

El método es del mismo tipo que los elementos de la lista.

resultado es del mismo tipo T.

Se puede llamar a suma() porque T es Operable.

Nota: No se puede realizar este método con comodines, porque el tipo T se necesita para el tipo del método y para declarar resultado.

Ejercicio 9.20:

Escriba un método elevaCuadrado() que modifique una lista de elementos de cualquier tipo que implemente Operable, de manera que se eleve al cuadrado cada elemento de la lista.

Planteamiento: La lista puede ser de cualquier tipo que implemente Operable, es decir puede que no sea una lista de Operable sino de Racionales.

Además el método devuelve un objeto del mismo tipo que los elementos de la lista. Por tanto hay que escribir un método genérico, que tiene como parámetro T extends Operable<T>.

El método se implementa recorriendo la lista y calculando el cuadrado de cada elemento, para ello se utiliza el método `producto()`. El resultado se vuelve a almacenar en esa misma posición utilizando el método `set()` de `List`.

Solución:

```
public static <T extends Operable<T>> void elevaCuadrado(List<T> lista){
   for (int i = 0; i < lista.size(); i++){
      T resultado = lista.get(i).producto(lista.get(i));
      lista.set(i,resultado);
   }
}
```

El método es del mismo tipo que los elementos de la lista.

`resultado` es del mismo tipo `T`.

Se puede llamar a `producto()` porque los elementos de la lista son `Operable`.

Nota: No se puede realizar este método con comodines, porque el tipo `T` se necesita para el tipo del método y para declarar resultado.

Ejercicio 9.21:

Escriba una aplicación donde se declare una lista de Racional y haga lo siguiente:

a) Iintroduzca 3 objetos de este tipo.
b) Imprima por pantalla la suma de todos los elementos de la lista utilizando el método sumaTodos().
c) Eleve al cuadrado todos los elementos de la lista utilizando el método elevaCuadrado().
d) Muestre por pantalla la lista para comprobar que se han modificado los elementos.

Nota: Suponga que los métodos utilizados están en la misma clase que el método `main()` de la aplicación.

Planteamiento: Simplemente hay que llamar a los métodos, al ser una lista de elementos que implementan la interfaz `Operable`, estos aceptarán la lista como parámetro. Para mostrar la lista se utiliza el método `toString()` al que no es necesario llamar explícitamente.

Solución:

```
   public static void main(String[] args) {
      Racional r1 = new Racional(1,2);
      Racional r2 = new Racional(2,3);
      Racional r3 = new Racional(1,4);

      List<Racional> racionales = new ArrayList<Racional>();

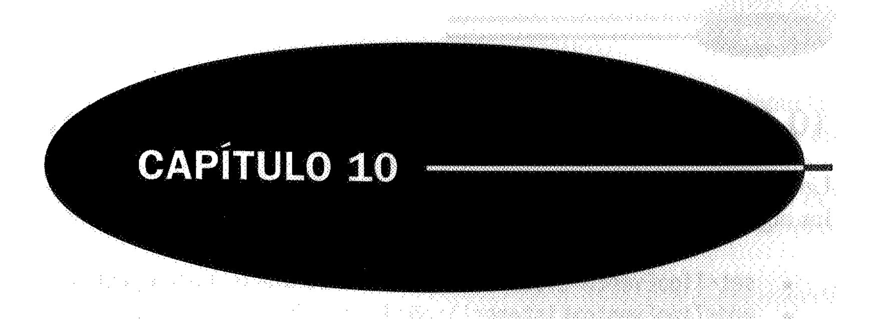
      racionales.add(r1);
      racionales.add(r2);
      racionales.add(r3);

      System.out.println(sumaTodos(racionales));
      System.out.println(racionales);
      elevaCuadrado(racionales);
      System.out.println(racionales);
   }
}
```

Se crea el `ArrayList` de tipo `Racional`.

Acepta la lista como parámetro porque almacena objetos `Operables`.

Nota: Si prueba a llamar a los métodos `sumaTodos()` y `elevaCuadrado()` con una lista de objetos que no sean `Operables`, se produce un error en tiempo de ejecución.



Interfaces gráficas de usuario con Swing

10.1 CREACIÓN DE UNA INTERFAZ GRÁFICA

Los pasos básicos para la creación de una interfaz gráfica de usuario son los siguientes:

1. El diseño y composición de la apariencia de la interfaz gráfica de la aplicación. Normalmente, esto implica la elección de una ventana principal (contenedor), la elección de contenedores para la jerarquía de componentes, la elección de los administradores de disposición para cada uno de los contenedores y los elementos gráficos de interacción (componentes primitivos).
2. Se escribe el código que crea todos los componentes, crea la jerarquía de componentes contenedores y componentes primitivos y da la apariencia a la interfaz.
3. Escribir el código que proporciona el comportamiento de dicha interfaz como respuesta a las interacciones de los usuarios para gestionar los eventos de interés para la aplicación.
4. La visualización de la interfaz gráfica. Una vez visualizada la interfaz, la aplicación queda a la espera de las interacciones del usuario que provocarán la ejecución de los gestores correspondientes.

10.2 TRATAMIENTO DE EVENTOS: EL MODELO DE DELEGACIÓN

Desde Java 1.1 el tratamiento de eventos se realiza mediante el **modelo de delegación de eventos** (*event model delegation*).

El tratamiento de un evento que ocurre en un objeto (*objeto fuente*) no se realiza en ese mismo objeto, sino que se delega en otro objeto diferente (*objeto oyente*). Este modelo se basa en la posibilidad de propagación o envío de eventos desde el objeto fuente donde se producen a los objetos oyentes que los gestionan. La forma de hacerlo es la siguiente:

- El objeto fuente registra qué objetos oyentes están interesados en recibir algún evento específico para comunicárselo una vez que éste se produzca.
- Cuando ocurre el evento, el objeto fuente se lo comunica a todos los oyentes registrados.
- La comunicación entre la fuente y el oyente se realiza mediante la invocación de un método del oyente al que se proporciona como argumento el evento generado.

10.2.1 Eventos, objetos fuente y objetos oyente

Un objeto fuente es aquel componente en el que se genera un evento. Para poder gestionar el registro y eliminación de los objetos oyentes de cada evento concreto se proporcionan los métodos:

- `set<TipoEvento>Listener()`: establece un único objeto oyente para ese tipo de evento.
- `add<TipoEvento>Listener()`: añade otro objeto oyente a la lista de oyentes.
- `remove<TipoEvento>Listener()`: elimina un objeto oyente correspondiente.

10.3 JERARQUÍA Y TIPOS DE EVENTOS

Conceptualmente los eventos generados en los componentes gráficos se pueden clasificar en: *eventos de bajo nivel* y *eventos semánticos* o de alto nivel. Los eventos de bajo nivel están relacionados con los aspectos físicos de la interacción con los elementos de la interfaz.. El resto de los eventos son de alto nivel. En la Tabla 10.1 se presentan los eventos más utilizados en Swing.

Tabla 10.1. Eventos más utilizados en Swing y su significado.

Eventos de bajo nivel	
Evento	**Significado**
`ComponentEvent`	Cambios en el tamaño, posición o visibilidad de un componente.
`FocusEvent`	Cambio de foco (capacidad de un componente para recibir entradas desde el teclado).
`KeyEvent`	Operación con el teclado.
`MouseEvent`	Operación con los botones del ratón o movimientos del ratón.
`WindowEvent`	Cambio de estado en una ventana.
`AncestorEvent`	Cambio en la composición, visibilidad o posición de un elemento superior (ancestro) de la jerarquía de composición.
Eventos de alto nivel	
`ActionEvent`	Realización de la acción específica asociada al componente.
`ChangeEvent`	Cambio en el estado del componente.
`ItemEvent`	Elemento seleccionado o deseleccionado.
`CaretEvent`	Cambio en la posición del cursor de inserción en un componente que gestiona texto.
`ListSelectionEvent`	Cambio en la selección actual en una lista.

10.4 CLASES OYENTES Y ADAPTADORAS DE EVENTOS

Los objetos que tratan los eventos se crean implementando las interfaces `<TipoEvento>Listener`. La mayoría de estas interfaces Java de los oyentes están diseñadas para responder a varios eventos diferentes, de modo que incluyen más de un método.

Por ejemplo, la interfaz oyente de interacciones del ratón `MouseInputListener` tiene siete métodos, tres relacionados con la operación del botón: `mousePressed()`, `mouseReleased()`, y `mouseClicked()`, y cuatro relacionados con el movimiento: `mouseEntered()`, `mouseExited()`, `mouseMoved()`, y `MouseDragged()`. Esto provoca que la clase oyente deba implementarlos todos, aunque sólo le interese alguno de ellos (por ejemplo, `mouseClicked`), y deje vacíos los otros, si no la clase sería abstracta y no se podrían crear objetos de la misma.

Para simplificar la escritura de oyentes, Java proporciona un conjunto de clases adaptadoras, que implementan las interfaces oyentes con todos los cuerpos de los métodos vacíos. Así un oyente se puede crear especializando un adaptador e implementando sólo el método que interese.

Utilizando la clase adaptadora `MouseInputAdapter` se puede escribir una clase oyente para las pulsaciones del ratón en un botón de la siguiente forma:

```
// oyente para las pulsaciones del ratón
class OyenteRaton extends javax.swing.event.MouseInputAdapter {
   public void mouseClicked (MouseEvent evento){
   // se obtiene el botón fuente del evento
   JButton boton = (JButton) evento.getSource();
   // se modifica la etiqueta con el nombre del botón pulsado, por ejemplo
   etiqueta.setText("Botón pulsado: " + boton.getText());
   }
}
```

Las clases adaptadoras simplifican el uso de clases internas anónimas para la gestión de eventos. En el siguiente ejemplo se escribe un oyente `WindowListener` que escribe código de atención al evento de desactivación de la ventana:

```
ventana.addWindowListener(new WindowAdapter(){
   public void windowDeactivated(WindowEvent e) {
      etiqueta.setText("Ventana desactivada");
   }
});
```

10.5 CONTENEDORES Y COMPONENTES EN JAVA

Los componentes o elementos gráficos para crear interfaces gráficas están divididos en dos grandes grupos: los contenedores y los componentes. Un componente, también denominado componente simple o atómico, es un objeto que tiene una representación gráfica, que se puede mostrar en la pantalla y con la que puede interactuar el usuario. Ejemplos de componentes son los botones, campos de texto, etiquetas o casillas de verificación. Un contenedor es componente al que se incluirán uno o más componentes o contenedores.

Los contenedores permiten generar la estructura de una ventana y el espacio donde se muestra el resto de los componentes contenidos en ella y que conforman la interfaz de usuario. Los contenedores de alto nivel más utilizados de Swing son:

- La clase `JFrame` proporciona una ventana principal de aplicación con su funcionalidad normal (por ejemplo, borde, título, menús) y un panel de contenido.
- La clase `JDialog` proporciona una ventana de diálogo auxiliar en una aplicación, normalmente utilizada para pedir datos al usuario o configurar elementos.
- La clase `JApplet` implementa una ventana que aparece dentro de otra aplicación que normalmente es un navegador de Internet (las *applets* se tratan en el Capítulo 11).

10.6 COMPONENTES GRÁFICOS: JERARQUÍA Y TIPOS

Las clases gráficas se presentan estructuradas en los siguientes grandes grupos:

- Clases básicas.
- Contenedores de alto nivel.

- Contenedores intermedios.
- Componentes atómicos.

10.6.1 Clases básicas

Proporcionan el soporte y las funcionalidades básicas para el resto de componentes. La raíz de todos los elementos gráficos en Java es la clase abstracta `java.awt.Component` y su subclase abstracta `java.awt.Container`. La clase `javax.swing.JComponent`, que es una especialización de `java.awt.Container`, es la clase base para prácticamente todos los componentes Swing. En su API se encuentran los métodos comunes que puede utilizar para todos los componentes y contenedores.

10.6.2 Contenedores de alto nivel

JFrame
`JFrame` se emplea para crear la ventana principal de una aplicación. Es una ventana con marco que incluye los controles habituales de cambio de tamaño y cierre (por ejemplo, cerrar, iconizar, maximizar).

A este contenedor se le puede añadir una barra de menús (`JMenuBar`). Los componentes gráficos no se añaden directamente al `JFrame` sino a su panel de contenido. De la misma manera el gestor de disposición, que aplica el diseño de presentación de los componentes, se debe aplicar a este panel de contenido. Los métodos más utilizados son:

```
f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); // cerrar la ventana
f.setLayout(objetoLayout); // establece el administrador de disposición
f.setTitle("Aplicación de prueba "); // pone un texto en el título de la ventana
f.setSize(300, 200); // establece el tamaño de la ventana
f.add(componente);   // añade un componente a la ventana
f.addXXXXXListener(objetoListener);   // añade un escuchador de eventos
f.setJMenuBar(barraDeMenu);  // añade una barra de menú a la ventana
f.setVisible(true);   // hace visible o invisible la ventana
f.pack();  // asigna tamaño iniciales a los componentes
```

JDialog
La clase `JDialog` es la clase raíz de las ventanas secundarias que implementan cuadros de diálogo en Swing. Estas ventanas dependen de una ventana principal (o con marco, normalmente de clase `JFrame`) y si la ventana principal se cierra, se iconiza o se desiconiza, las ventanas secundarias realizan la misma operación de forma automática. Estas ventanas pueden ser modales o no modales, es decir, limitan la interacción con la ventana principal si así se desea. El constructor más utilizado es:

```
// crea un JDialog con la ventana de la que depende, un título y si es modal o no.
JDialog dialogo = new JDialog(frame, "Título", true);
```

Se pueden utilizar todos los métodos descritos para la clase `JFrame`.

10.6.3 Cuadros de diálogo estándar

JOptionPane
Esta clase se utiliza para crear los tipos de cuadros de diálogo más habituales, como los que permiten pedir un valor, mostrar un mensaje de error o advertencia, solicitar una confirmación, etc. Todos los cuadros de diálogo que implementa

`JOptionPane` son modales (es decir bloquean la interacción del usuario con otras ventanas). La forma habitual de uso de la clase es mediante la invocación de alguno de sus métodos estáticos para crear cuadros de diálogo. Tienen el formato `show<Tipocuadro>Dialog`, donde el tipo de cuadro puede ser:

a) *Message* para informar al usuario con un mensaje.
b) *Confirm* para solicitar una confirmación al usuario con las posibles respuestas de si, no o cancelar.
c) *Input* para solicitar la entrada de un dato.
d) *Option* permite crear una ventana personalizada de cualquiera de los tipos anteriores

Las formas más habituales de uso se pueden resumir en las siguientes:

```
JOptionPane.showMessageDialog(ventana,
        "No se ha podido encontrar el archivo indicado",  // mensaje
        "Error de entrada de datos",  //título
        JOptionPane.ERROR_MESSAGE);   //icono

// cuadro de confirmación. Devuelve YES_OPTION, NO_OPTION
int opcion = JOptionPane.showConfirmDialog(ventana,
        "¿Desea realmente salir?",  // mensaje
        "Seleccione si desea salir",  //título
        JOptionPane.YES_NO_OPTION);

// cuadro de confirmación. Devuelve el texto introducido o null si se cancela
String entrada = JOptionPane.showInputDialog(ventana,
        "Introduzca un número entero:",  // mensaje
        "Solicitud de un número",  //título
        JOptionPane.QUESTION_MESSAGE);

// cuadro de opción personalizado, devuelve en n el botón pulsado
Object[] opciones = {"Sí, adelante", "Ahora no", "No sé que hacer"};
int n = JOptionPane.showOptionDialog(ventana,
        "¿Desea realizar la operación ahora?", // mensaje
        "Mensaje de confirmación", //título
        JOptionPane.YES_NO_CANCEL_OPTION,
        JOptionPane.QUESTION_MESSAGE,
        null,      //utiliza el icono predeterminado
        textoOpciones,
        textoOpciones[0]); //botón predeterminado
```

JFileChooser
Se trata de un selector de archivos que permite la elección interactiva de un archivo o un directorio. Un uso habitual puede ser el siguiente:

```
// se crea el selector de ficheros
JFileChooser selector = new JFileChooser();
// solo posibilidad de seleccionar directorios
selector.setFileSelectionMode(JFileChooser.DIRECTORIES_ONLY);
// se muestra y se espera a que el usuario acepte o cancele la selección
int opcion = selector.showOpenDialog(ventana);
if (opcion == JFileChooser.APPROVE_OPTION) {
   // se obtiene el fichero o directorio seleccionado
```

```
        File archivo = selector.getSelectedFile();
    }
```

Otros métodos para el cuadro de diálogo selector de archivos son:

```
int opcion = selector.showSaveDialog(ventana); // para guardar
selector.setCurrentDirectory(directorio); // pone el directorio inicial
File[] archivos = selector.getSelectedFiles(); // si admite selección múltiple
```

JColorChooser
Sirve para elegir un color de una paleta que presenta en cuadro de diálogo. El uso básico es el siguiente:

```
Color color = JColorChooser.showDialog(ventana,
                 "Elija el color deseado", // título de la ventana
                 Color.WHITE); // color inicial
```

10.6.4 Contenedores intermedios

JPanel
JPanel es un contenedor simple de propósito general que sirve para agrupar a otros componentes. Habitualmente se utiliza para agrupar componentes a los que se aplica un gestor de disposición adecuado. Permiten añadirles bordes o personalizar su presentación gráfica. Los métodos más utilizados son los siguientes:

```
JPanel panel = new JPanel();       // o new JPanel(objetoLayout)
panel.setLayout(objetoLayout);     // establece el administrador de disposición
panel.add(componente);             // añade un componente al panel
panel.setBorder(objetoBorde);      // pone un borde al panel
panel.setTooltipText(texto);       // crea una ayuda que aparece al dejar el ratón
```

JScrollPane
La clase JScrollPane proporciona un panel con la capacidad de presentar barras de desplazamiento para mostrar su contenido. Es adecuado para presentar información que no cabe completamente en la zona de visualización asignada. El componente que proporciona el contenido se denomina cliente y la zona de presentación de información se denomina puerto de visualización.

Además del puerto de visualización y de las barras de desplazamiento JScrollPane tiene otras zonas que son una cabecera horizontal, otra vertical y las cuatro esquinas. Todos estos elementos son configurables. La forma de uso más habitual incluye los siguientes métodos:

```
// las barras de desplazamiento aparecen sólo si son necesarias
JScrollPane scroll = new JScrollPane(componente);
// las barras de desplazamiento siempre presentes
JScrollPane scroll = new JScrollPane(componente,
                 JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS,
                 JScrollPane.VERTICAL_SCROLLBAR_ALWAYS);
scroll.setBorder(objetoBorde);  // pone un borde al panel
scroll.setTooltipText(texto);   // crea una ayuda que aparece al dejar el ratón
```

JSplitPane
Es un contenedor que gestiona dos componentes (normalmente paneles) colocados vertical u horizontalmente y diferenciados por un separador que puede reposicionar el usuario. El usuario puede decidir cuál es el tamaño asignado a cada

uno de los componentes pulsando con el ratón sobre el separador y arrastrándolo. De modo predeterminado, se dividen en horizontal, aunque se puede establecer en el constructor. La forma más habitual de uso es la siguiente:

```
// la orientación también puede ser VERTICAL_SPLIT
JSplitPane panelDividido = new JSplitPane(JSplitPane.HORIZONTAL_SPLIT,
                          panelIzquierdo, panelDerecho);
panelDividido.setDividerLocation(200); // en pixel
panelDividido.setDividerLocation(30.0); // en porcentaje
panelDividido.setHorientation(VERTICAL_SPLIT); // modifica la orientación
panelDividido.setBorder(objetoBorde); // pone un borde al panel
panelDividido.setTooltipText(texto);  // crea una ayuda que aparece al dejar el ratón
```

JTabbedPane
El panel con solapas es un contenedor que gestiona varios grupos de componentes como una pila de fichas. Los componentes se superponen unos a otros de forma que sólo uno de ellos es visible en cada momento. El usuario decide cuál de los componentes se visualiza seleccionando la solapa o lengüeta correspondiente a dicho componente. Las solapas incluyen un texto y/o un icono y se pueden colocar en las partes superior, inferior, derecha o izquierda del contenedor. El administrador de disposición por defecto de este componente es `CardLayout`. La forma más habitual de uso y los métodos más utilizados son los siguientes:

```
// crea un TabbedPane con las solapas a la izquierda
JTabbedPane panelSolapas = new JTabbedPane(JTabbedPane.LEFT);
// añade una pestaña con icono y tooltip
panelSolapas.addTab("Primero", icono, contenedor, "Texto tooltip");
// añade una pestaña con icono y tooltip en una determina posición
panelSolapas.insertTab("Otro", icono, contenedor, "Texto tooltip", 3);
panelSolapas.getSelectedIndex(); // devuelve el índice de la pestaña seleccionada
// si no caben las pestañas que las ponga en otra fila
panelSolapas.setTabLayoutPolicy(JTabbedPane.WRAP_TAB_LAYOUT);
panelSolapas.setTabPlacemen(JTabbedPane.BOTTOM); // pone las pestañas abajo
```

JToolBar
Esta clase implementa una barra de herramientas, formada normalmente por botones o controles que incluyen iconos, y que aparecen organizados como una fila o una columna dependiendo de la zona de la pantalla donde se coloque. Por defecto, el usuario puede seleccionar y desplazar esta barra de herramientas a cualquier borde del contenedor que la incluye (se recomienda usar el administrador de disposición `BorderLayout`) o a una ventana independiente. Si el programador no quiere que se pueda desplazar, tiene que desactivar dicha posibilidad mediante `barraHerramientas.setFloatable(false)`. El administrador de disposición por defecto de una barra de herramientas es `BoxLayout`. El uso y métodos más habituales de este componente son los siguientes:

```
JToolBar barra = new JToolBar();
barra.add(componente); // añade un componente a la barra, suele ser un botón.
barra.addSeparator(); // añade un separador
barra.setFloatable(false); // impide que se pueda mover de sitio
barra.setOrientation(JToolBar.VERTICAL); // pone la orientación vertical
```

10.6.5 Componentes atómicos

Iconos
Un icono es una imagen gráfica (pequeña y de tamaño fijo) que habitualmente se utiliza para presentarla dentro de otro componente, como un botón, una etiqueta u otros. Puede crearse a partir de imágenes .gif o .jpg de la siguiente forma:

```
ImageIcon icono1 = new ImageIcon("imagen.gif"); // con un nombre de archivo
ImageIcon icono2 = new ImageIcon(urlDeLaImagen); // con una URL al archivo
```

JLabel

Esta clase implementa una etiqueta que puede contener una cadena de texto, un icono o ambos. En una etiqueta se puede especificar donde aparece su contenido indicando el alineamiento vertical y horizontal. Por defecto, las etiquetas se muestran centradas verticalmente, y si sólo tienen texto, ajustadas a la izquierda. Si sólo tienen una imagen gráfica por defecto se muestran centradas tanto vertical como horizontalmente. Es posible indicar la posición relativa del texto con relación al icono. También se puede utilizar como texto un fragmento de HTML. Un uso habitual de un `JLabel` es el siguiente:

```
JLabel etiq = new JLabel("Texto", icono, JLabel.RIGHT); // con icono
JLabel etiq = new JLabel("Texto de la etiqueta"); // solo con texto
etiq.setText("Nuevo texto de la etiqueta"); // cambia el texto
etiq.setIcon(nuevoIcono); // cambia el icono
etiq.setText("<html>Nuevo <i>texto</i> de la etiqueta</html>"); // pone texto html
etiq.setHorizontalAlignement(JLabel.LEFT); // cambia la alineación
etiq.setHorizontalTextPosition(JLabel.LEFT); // cambia la posición del texto
etiq.setVerticalAlignement(JLabel.TOP); // cambia la alineación
etiq.setVerticalTextPosition(JLabel.TOP); // cambia la posición del texto
```

JButton

Esta clase implementa la forma más habitual de botón gráfico de interacción que sirve para ejecutar una acción haciendo clic sobre él. También se puede activar mediante el teclado si se le ha asociado una combinación de teclas. Puede tener un texto y/o un icono.

```
JButton boton1 = new JButton("Aceptar", icono); // botón con texto e icono
JButton boton2 = new JButton("Cancelar"); // botón solo con texto
boton1.setMnemonic(KeyEvent.VK_C); // asigna como atajo de teclado Alt+C
boton1.setText("<html>Botón con <p> texto <b>html</b>"); // con texto html
boton1.setTooltipText("Botón de aceptación"); // una ayuda textual
frame.getRootPane().setDefaultButton(boton1); // pone como botón predeterminado
```

JToggleButton

Esta clase implementa un botón de operación que tiene un estado interno y funciona como un conmutador. Cuando se pulsa el botón su estado pasa a estar seleccionado hasta que se vuelve a pulsar de nuevo, en cuyo caso deja de estar seleccionado. El uso y métodos más habituales son los siguientes:

```
JToggleButton boton1 = new JToggleButton("Aceptar", icono); // con texto e icono
JToggleButton boton2 = new JToggleButton("Cancelar", false); // botón deseleccionado
boton1.setMnemonic(KeyEvent.VK_A); // asigna como atajo de teclado Alt+A
boton1.setText("<html>Botón con <p> texto <b>html</b>"); // con texto html
boton1.setTooltipText("Botón de aceptación"); // una ayuda textual
boton1.setSelected(false); // deselecciona el botón
frame.getRootPane().setDefaultButton(boton1); // pone como botón predeterminado
```

JCheckBox

Es una casilla de verificación con dos estados posibles: seleccionada o no seleccionada, que determina y modifica su apariencia gráfica (normalmente mediante una cruz o marca de selección). Generalmente se utiliza para permitir que el usuario decida si desea elegir una opción o no. Si hay varias casillas de verificación, éstas no son mutuamente excluyentes, de modo que varias de ellas pueden estar seleccionadas de forma simultánea. El uso y métodos más habituales son los siguientes:

```
JCheckBox casilla1 = new JCheckBox("Manzana", icono); // con texto e icono
JCheckBox casilla2 = new JCheckBox("Naranja", true); // casilla seleccionada
casilla1.setMnemonic(KeyEvent.VK_M); // asigna como atajo de teclado Alt+M
casilla1.setTooltipText("Naranja como fruta"); // una ayuda textual
casilla1.setSelected(false); // deselecciona la casilla
```

Los elementos que tienen este mismo comportamiento pero que aparecen dentro de un menú están implementados en la clase JCheckBoxMenuItem.

JRadioButton
Esta clase implementa los botones de radio o de opción que son una especialización de un botón con estado o conmutador y se caracterizan porque en un grupo de botones de radio sólo uno de ellos puede estar seleccionado. Para crear un grupo de botones de radio hay que añadirlos en un objeto ButtonGroup, que no tiene representación gráfica. El uso y métodos más habituales son los siguientes:

```
JRadioButton boton1 = new JRadioButton("Manzana", icono); // con texto e icono
JRadioButton boton2 = new JRadioButton("Naranja", false); // botón deseleccionado
boton1.setMnemonic(KeyEvent.VK_M); // asigna como atajo de teclado Alt+M
boton1.setTooltipText("Naranja como fruta"); // una ayuda textual
boton1.setSelected(false); // deselecciona el botón de radio

// se agrupan en un ButtonGroup
ButtonGroup grupo = new ButtonGroup();
grupo.add(boton1);
grupo.add(boton2);
```

Los elementos que tienen el mismo comportamiento, pero que aparecen dentro de un menú, están implementados en la clase JRadioButtonMenuItem.

JMenuBar, JMenu, JMenuItem y otros elementos
Implementa una barra de menús que se añade a un contenedor de alto nivel y, posteriormente, se le añaden los menús desplegables (JMenu).

Las barras de menú están implementadas por la clase JMenuBar. La clase JPopupMenu implementa los menús contextuales o emergentes.

A estos dos tipos de menús se le puede añadir cualquiera de los elementos de menú (JMenuItem) o sus especializaciones, que son los menús desplegables (JMenu), o las particularizaciones de las casillas de verificación (JCheckBoxMenuItem) y de los botones de radio (JRadioButtonMenuItem) para los menús. Para facilitar la agrupación lógica de las operaciones incluidas en un menú, además, se pueden incluir separadores de menú.

Las casillas de verificación (JCheckBoxMenuItem) y los botones de radio (JRadioButtonMenuItem) de los menús funcionan de la misma forma y tienen métodos similares que los botones correspondientes (JCheckBox y JRadioButton).

El uso y métodos más habituales son los siguientes:

```
JMenuBar barraMenu = new JMenuBar(); // crea la barra de menús
JMenu menuOpciones = new JMenu("Menú de opciones"); // crea un nuevo menú
JMenuItem listar = new JMenuItem("Listar todos los alumnos"); // crea un elemento
menuOpciones.add(listar);
menuOpciones.addSeparator; // añade un separador
JMenuItem listarTarde = new JMenuItem("Ver alumnos de la tarde");
menuOpciones.add(listarTarde);
barraMenu.add(menuOpciones);
frame.setJMenuBar(barraMenu); // añade el menú a la ventana principal
```

JTextField

Componente que permite mostrar y editar una única línea de texto. Una especialización de los campos de texto son los campos de contraseña (JPasswordField) que tienen la particularidad de que no muestran el contenido que se escribe sino otro carácter (por ejemplo, un asterisco) o los campos de texto con formato (JFormattedTextField) que permiten especificar el formato de texto que acepta. El uso y métodos más habituales son los siguientes:

```
JTextField texto = new JTextField("texto inicial", 10); // crea el campo con 10 colum
String s = texto.getText(); // obtiene el texto del campo
texto.setText("nuevo texto"); // pone un nuevo texto

JPasswordField pass = new JTextField(10); // crea el campo con 10 columnas
char[] contraseña = texto.getPassword(); // obtiene la contraseña
```

JTextArea

Este componente permite mostrar y editar varías líneas de texto. Su limitación principal es que sólo permite mostrar texto sencillo en el que se utilice un único tipo de letra. Habitualmente se utiliza asociado a un panel con desplazamiento. El uso y métodos más habituales son los siguientes:

```
JTextArea areaTexto = new JTextArea(10, 20); // crea con 10 filas y 20 columnas
texto.setLineWrap(true); // corta el texto al final del área
texto.setWrapStyleWord(true);   // corta el texto por palabras
texto.append("texto añadido"); // añade texto al final del texto
texto.insert("texto inicial", 15); // inserta un texto en la posición 15
```

JEditorPane

Esta clase proporciona soporte para campos de texto de múltiples líneas y con formato. Además de texto simple soportan texto en formato HTML y texto en formato enriquecido (RTF, *Rich Text Format*). Aunque son básicamente editores de texto sencillos que implementan el ajuste de líneas, estos campos de texto no incluyen barras de desplazamiento y, por tanto, normalmente se utilizan asociados a un panel con desplazamiento.

JList

La clase JList implementa una lista de elementos que se presenta habitualmente en forma de columna. En esta lista el usuario puede realizar la selección de uno (comportamiento por defecto) o varios de sus elementos. Si la lista tiene muchos elementos y, por tanto, puede no caber en el espacio de visualización asignado debería incluirse en un panel con barras de desplazamiento. El uso y métodos más habituales son los siguientes:

```
String[] datos = {"Manzana", "Naranja", "Pera", "Melocotón", "Uva"};
JList lista = new JList(datos);
lista.addElement("Elemento añadido");
// permite seleccionar un intervalo de datos
lista.setSelectionMode(ListSelectionModel.SINGLE_INTERVAL_SELECTION);
// se cambia la orientación de presentación y el ajuste
lista.setLayoutOrientation(JList.HORIZONTAL_WRAP);
Object[] selección = lista.getSelectedValues(); // recoge los seleccionados
int[] indices = lista.getSelectedIndices(); //recoge los índices de los seleccionados
// se añade a un JScrollPane para que tenga barras de desplazamiento.
JScrollPane panelDesplazamiento = new JScrollPane(lista);
```

JComboBox

Esta clase implementa un cuadro combinado desplegable, en el que se agrupan las funcionalidades de una lista y un campo de texto. Cuando el usuario selecciona un elemento de la lista, ese valor pasa automáticamente a mostrarse

como contenido del cuadro combinado. Tiene dos posibles formas de uso. La primera y más simple en la que sólo se permite que el usuario escoja una opción de entre una lista fija de posibles valores. La segunda forma de uso es en la que el usuario también puede introducir un nuevo valor. El uso y métodos más habituales son los siguientes:

```
String[] datos = {"Manzana", "Naranja", "Pera", "Melocotón", "Uva"};
JComboBox combo1 = new JComboBox(datos);
combo1.addItem("Kiwi"); // añade un elemento más
combo1.setSelectedItem("Pera"); // aparece como seleccionado
combo1.setEditable(true); // permite escribir valores
combo1.setMaximumRowCount(5); // se muestran sólo 5 valores
int n = combo1.getSelectedIndex(); // devuelve el índice del seleccionado
Object o = combo1.getSelectedItem(); // devuelve el elemento seleccionado
combo1.removeItemAt(3); // elimina el tercer elemento
```

10.6.6 Otras clases gráficas de Swing

Swing incorpora otros objetos gráficos. Los más utilizados se enumeran a continuación. Si desea más información consulte la API de Java.

- JProgressBar. Barra configurable que muestra de forma gráfica la progresión temporal de una operación como un porcentaje de la longitud de la barra.
- JScrollBar. Barra de desplazamiento.
- JSlider. Barra gráfica con un indicador deslizante asociado que sirve para obtener datos de entrada proporcionados por el usuario.
- JTable. Componente altamente configurable que permite visualizar tablas bidimensionales.
- JToolTip. Componente que muestra en una ventana emergente una breve información de ayuda contextual sobre otros componentes cuando el cursor se sitúa sobre ellos.
- JTree. Componente que permite visualizar datos organizados jerárquicamente en forma de árbol.

10.7 ADMINISTRADORES DE DISPOSICIÓN O DISEÑO (LAYOUT MANAGERS)

10.7.1 FlowLayout

Es el administrador de disposición mas simple que proporciona Java y es el que se proporciona por defecto en los paneles (JPanel). Los componentes se colocan de forma secuencial, uno a continuación de otro, hasta ocupar el espacio completo asignado al contenedor según el orden en el que han sido añadidos. Si no caben en una línea se sigue en la siguiente. La dirección de colocación de los componentes depende de la propiedad componentOrientation del contenedor cuyo valor puede ser LEFT_TO_RIGHT o RIGHT_TO_LEFT.

10.7.2 BoxLayout

Con BoxLayout los componentes se organizan en una única fila, o en una única columna. La ordenación concreta dentro de la fila o columna puede ser absoluta: X_AXIS (horizontalmente de izquierda a derecha) e Y_AXIS (verticalmente de arriba abajo), o relativa dependiendo de la característica de orientación del componente: LINE_AXIS (como se organizan las palabras en una línea) y PAGE_AXIS (como se organizan las líneas en una página). Es el diseño por defecto de las barras de herramientas (JToolBar).

En el constructor hay que indicar el contenedor al que se va a asignar. Se suele hacer en un paso de la siguiente forma cambiando el Layout a un objetoContenedor: `objetoContenedor.setLayout(new BoxLayout(objetoContenedor, Y_AXIS));`

Además de componentes primitivos permite incluir elementos invisibles que crean separaciones entre los otros componentes del contenedor. Los elementos invisibles más útiles son las áreas rígidas y los "pegamentos o gomas extensibles". Las áreas rígidas (`Box.createRigidArea()`) proporcionan un espacio fijo entre dos componentes. Si se añade un objeto "pegamento o goma extensible" (`Box.createGlue()`) intenta ocupar todo el espacio posible. Si hay varios se reparten el espacio entre ellos por igual.

10.7.3 BorderLayout

Este administrador de disposición está basado en dividir el contenedor en cinco zonas, una central y otras cuatro según los puntos cardinales. La zona central trata de ocupar la mayor parte posible del espacio del contenedor. Es el administrador por defecto en el panel de contenido de los contenedores de alto nivel (por ejemplo, `JFrame`, `JDialog`). Al añadir un componente hay que indicar la zona en que se quiere colocar. Cada una de las zonas está identificada por la correspondiente constante de clase: `CENTER`, `NORTH`, `SOUTH`, `EAST` y `WEST`.

10.7.4 CardLayout

Permite gestionar distintos componentes (normalmente paneles) que ocupan un mismo espacio, de forma que en cada momento sólo uno de ellos es visible. Utiliza la idea de un mazo de tarjetas o cartas donde sólo es visible la primera de ellas. La ordenación del mazo es el orden en el que se han añadido los componentes. `CardLayout` define un conjunto de métodos que permiten recorrer el mazo de cartas secuencialmente (de la primera a la última o al revés) o mostrar una carta determinada a partir de su nombre.

10.7.5 GridLayout

Con `GridLayout` los componentes se colocan en una matriz de celdas, definida por filas y columnas, que ocupa todo el espacio asignado al contenedor. Todas las celdas son iguales y los componentes utilizan todo el espacio disponible para cada celda. Al añadir los componentes se va rellenando la matriz primero por filas (en función de la orientación del componente) y cuando una fila esta llena se pasa a la siguiente.

10.7.6 GridBagLayout

Permite organizar los componentes tanto verticalmente como horizontalmente aunque tengan distinto tamaño. Con este propósito mantiene una matriz de celdas, donde las filas y columnas pueden tener distinto tamaño, en las que se sitúan los componentes. Un componente puede ocupar varias celdas contiguas. La situación y el espacio ocupado por cada uno de los componentes se describen por medio de restricciones (*constrains*). A cada componente se le asocia un objeto de la clase `GridBagConstrains` que especifica las restricciones y que, junto con las indicaciones de tamaño del componente (tamaño mínimo y tamaño preferido), determinan la presentación final del objeto en el contenedor.



Problemas resueltos

INTERFACES GRÁFICAS DE USUARIO

Ejercicio 10.1:

Escriba una aplicación con interfaz gráfica en la que se construya una ventana con título y marco que tenga los controles básicos (iconizar, maximizar, cerrar) y que al pulsar sobre el aspa de la ventana (cerrar) se salga completamente de la aplicación. La ventana contendrá una etiqueta y el usuario debe poder cambiar su tamaño.

Planteamiento:

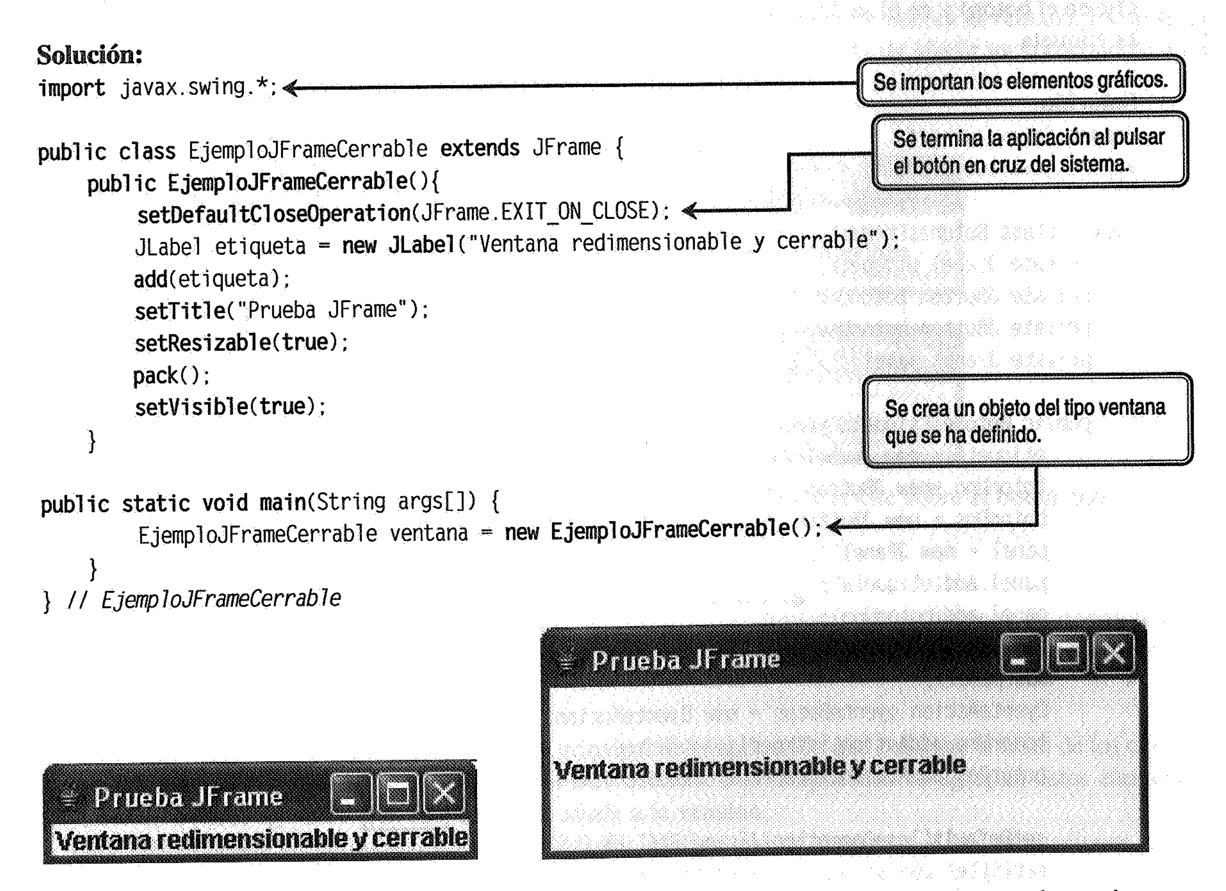
a) Como hay que crear una ventana con marco lo mas adecuado es crear una clase que extienda o especialice la clase `JFrame`. Esta clase ya tiene marco y los controles básicos.

b) Sólo hay que configurar el comportamiento del control de cierre (el aspa) de la ventana. Para hacerlo simplemente se fija la propiedad correspondiente mediante la instrucción `setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE)`.

c) Hay que crear una etiqueta de tipo `JLabel` y añadirla a la ventana.

d) Se configuran el resto de propiedades de la ventana, como son su título, que se pueda modificar su tamaño, que su tamaño inicial se ajuste al tamaño preferido de los componentes que contiene y que sea visible.

Solución:



```
import javax.swing.*;

public class EjemploJFrameCerrable extends JFrame {
    public EjemploJFrameCerrable(){
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        JLabel etiqueta = new JLabel("Ventana redimensionable y cerrable");
        add(etiqueta);
        setTitle("Prueba JFrame");
        setResizable(true);
        pack();
        setVisible(true);
    }

public static void main(String args[]) {
        EjemploJFrameCerrable ventana = new EjemploJFrameCerrable();
    }
} // EjemploJFrameCerrable
```


Figura 10.1. Captura de la ventana generada inicialmente y una vez redimensionada por el usuario.

Comentario: El comportamiento de cierre de la aplicación se podría haber proporcionado mediante un gestor de eventos de ventana (un oyente `WindowListener`) que implemente el método `windowClosing()` en el cual se ejecutaría la operación de salida de la aplicación (`System.exit(0)`).

Aviso: En versiones de Java anteriores a la 1.4 los componentes gráficos no se podían añadir directamente la etiqueta a la ventana mediante `add(etiqueta)`. Había que hacerlo expresamente a su panel de contenido mediante `getContentPane().add(etiqueta)`.

Ejercicio 10.2:

Escriba una aplicación gráfica con una ventana que tenga una etiqueta y dos botones de operación. El comportamiento de la aplicación debe reflejar en el texto de la etiqueta cuál es el último botón en el que el usuario ha hecho clic.

Planteamiento: Primero se construye la ventana principal de la aplicación que contiene al resto de los componentes gráficos, es decir, la etiqueta y los dos botones. Por simplicidad se agrupa la etiqueta y los botones en un panel que es el que finalmente se añade a la ventana.

En este caso hay que proporcionar un comportamiento a la interfaz gráfica que dé respuesta a las interacciones de los usuarios, en concreto a los botones, y por tanto hay que incluir un manejador o gestor de eventos. Con el gestor se va a escuchar y responder al evento de alto nivel de acción que se genera al hacer clic sobre cada botón.

Este oyente se lleva a cabo como una clase interna que implementa la interfaz `ActionListener` y proporciona el método `actionPerformed()`. Este es el método que se ejecuta cuando se produce el evento (se hace clic en el botón) y en él se determina el origen del evento y, en función de cuál se haya pulsado, se modifica la etiqueta.

Solución:

```
import javax.swing.*;
import java.awt.event.*;   <-- Además de los elementos gráficos hay que importar los eventos.

public class BotonesEtiquetaOyente extends JFrame {
    private JLabel etiqueta;
    private JButton botonUno;
    private JButton botonDos;
    private JPanel panel;

    public BotonesEtiquetaOyente() {
        etiqueta = new JLabel("No se ha pulsado ningún botón");
        botonUno =new JButton("Boton Uno");
        botonDos = new JButton("Boton Dos");
        panel = new JPanel();
        panel.add(etiqueta);   <-- Se añaden los objetos gráficos al panel y el panel a la ventana.
        panel.add(botonUno);
        panel.add(botonDos);
        add(panel);
        OyenteAccion oyenteBoton = new OyenteAccion();   <-- Se crea un objeto oyente de acciones.
        botonUno.addActionListener(oyenteBoton);
        botonDos.addActionListener(oyenteBoton);   <-- Se asigna el mismo oyente a los dos botones de interacción.

        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setTitle("Ventana con interacción");
        pack();
```

```
        setVisible(true);
    }
    public static void main(String[] args) {
        BotonesEtiquetaOyente ventana = new BotonesEtiquetaOyente();
    }
    class OyenteAccion implements ActionListener{ ←— Se crea una clase oyente de acciones como clase interna.
        public void actionPerformed (ActionEvent evento){
            JButton boton = (JButton) evento.getSource();
            etiqueta.setText("Boton pulsado: " + boton.getText());
        }
    }// OyenteAccion
}//BotonesEtiquetaOyente
```

Comentario: En este caso el tratamiento de los eventos de acción generados por el clic del usuario sobre los botones se trata mediante un único objeto de una clase oyente interna a la clase principal de la aplicación. La ventaja de las clases internas es que pueden acceder directamente a todos los atributos de la clase contenedora y además, como son locales a ellas, se evita la proliferación de clases. Otras opciones posibles serían que la propia clase principal pudiera actuar como oyente de acciones, es decir, que implemente la interfaz `ActionListener` y proporcione el método `actionPerformed()` o implementar la clase oyente como una clase externa normal.

Aviso: Un componente gráfico sólo puede estar contenido en un único elemento gráfico contenedor. En este caso los botones y la etiqueta están contenidos en el panel y el panel está contenido en la ventana. Es decir, la jerarquía de composición de objetos gráficos es simple y, por tanto, si se trata de añadir un componente a un segundo contenedor, dejaría de estar contenido en el primer contenedor.

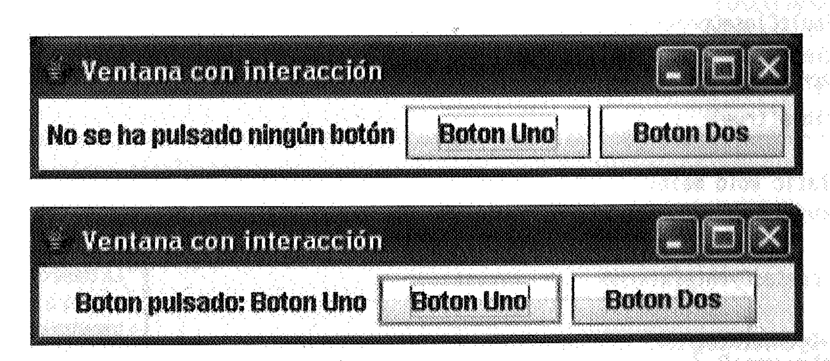


Figura 10.2. Ventana inicial de la aplicación y resultado después de hacer clic sobre el botón uno.

Ejercicio 10.3:

Escriba una aplicación gráfica con una ventana que tenga dos etiquetas y dos botones de operación. El comportamiento de la aplicación debe reflejar en el texto de las etiqueta el número de veces que el usuario ha hecho clic en cada uno de los botones.

Planteamiento: Primero se construye la ventana principal de la aplicación que contiene al resto de los componentes gráficos, es decir, las dos etiquetas y los dos botones. Por simplicidad se agrupan estos elementos gráficos en un panel que es el que finalmente se añade a la ventana.

Para proporcionar un comportamiento a la interfaz gráfica que dé respuesta a las interacciones de los usuarios, en concreto a los botones, hay que incluir un gestor de eventos que responda al evento de la acción que se genera al hacer clic sobre cada botón.

Solución:

```
import javax.swing.*;
import java.awt.event.*;

public class BotonesEtiquetasOyenteExterno extends JFrame {
    private JLabel etiquetaUno;
    private JLabel etiquetaDos;
    private JButton botonUno;
    private JButton botonDos;
    private JPanel panel;

    public BotonesEtiquetasOyenteExterno() {
      etiquetaUno = new JLabel("Boton Uno: 0 veces");
      etiquetaDos = new JLabel("Boton Dos: 0 veces");
      botonUno = new JButton("Boton Uno");
      botonDos = new JButton("Boton Dos");
      panel = new JPanel();
      panel.add(etiquetaUno);
      panel.add(botonUno);
      panel.add(etiquetaDos);
      panel.add(botonDos);
      add(panel);
      OyenteExternoAccion oyenteBotonUno = new OyenteExternoAccion(etiquetaUno);
      OyenteExternoAccion oyenteBotonDos = new OyenteExternoAccion(etiquetaDos);
      botonUno.addActionListener(oyenteBotonUno);
      botonDos.addActionListener(oyenteBotonDos);
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      setTitle("Ventana con interacción");
      setSize(300,100);
      setVisible(true);
    }
    public static void main(String[] args) {
        BotonesEtiquetasOyenteExterno ventana = new BotonesEtiquetasOyenteExterno();
    }
}//BotonesEtiquetasOyenteExterno

public class OyenteExternoAccion implements ActionListener{
    private int numVeces;
    private JLabel etiqueta;

    public OyenteExternoAccion(JLabel etiqueta){
        numVeces = 0;
        this.etiqueta = etiqueta;
    }

public void actionPerformed (ActionEvent evento){
            numVeces++;
            JButton boton = (JButton) evento.getSource();
            etiqueta.setText(boton.getText()+": "+ numVeces+ " veces");
        }
}// OyenteExternoAccion
```

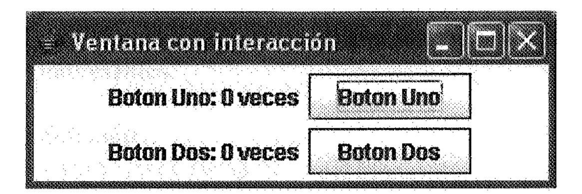
> Los componentes se presentan en el panel en el orden en el que se añaden.

> En este caso se crean dos objetos oyentes y se les pasa como parámetro la etiqueta correspondiente.

> La clase oyente es una clase pública externa a la clase en la que se generan los eventos.

> El oyente tiene un constructor que recibe como parámetro la etiqueta a modificar.

Comentario: Aunque las etiquetas son atributos privados de la ventana principal y el oyente de eventos es una clase externa a dicha ventana, no es necesario hacer un método de acceso ya que el constructor del oyente recibe como parámetro la etiqueta que tiene que modificar.



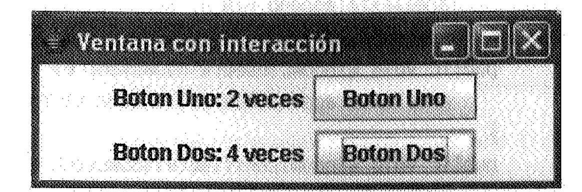


Figura 10.3. Ventana inicial de la aplicación y después de hacer clic dos veces sobre el botón uno y 4 veces sobre el botón dos.

Ejercicio 10.4:

Escriba una aplicación gráfica con una ventana que tenga una etiqueta y un área de texto. La aplicación debe reflejar en el área de texto todos los eventos de ventana que se produzcan por la creación de la ventana o por las interacciones del usuario.

Planteamiento: Primero se construye la ventana principal de la aplicación que contiene un panel en el que se incluyen la etiqueta y el área de texto. Para poder reflejar todos los eventos de la ventana se hace que la propia ventana implemente la interfaz de oyente de ventana y se proporcionan los siete métodos correspondientes de esta interfaz que añaden al área de texto una línea de texto indicando el evento producido.

Solución:

```
import javax.swing.*;
import java.awt.event.WindowListener;
import java.awt.event.WindowEvent;

public class VentanaOyente extends JFrame implements WindowListener {
    private JTextArea areaTexto;
    private JLabel etiqueta;
    private JPanel panel;
    private final String FIN ="\n";

   public VentanaOyente() {
        etiqueta  = new JLabel("Eventos");
        areaTexto = new JTextArea(10, 30);
        areaTexto.setText("Texto inicial del area de texto");
        panel = new JPanel();

      panel.add(etiqueta);
      panel.add(areaTexto);
      add(panel);
      addWindowListener(this);
      setDefaultCloseOperation(WindowConstants.DO_NOTHING_ON_CLOSE);
      setTitle("Ventana oyente");
      setVisible(true);
      pack();
   }
```

> La propia ventana gráfica es oyente de eventos de ventana ya que implementa la interfaz `WindowListener`.

> Se crea un área de texto con 10 filas y 30 columnas y se le asocia una cadena de texto inicial.

> Con `this` es la propia ventana la que se registra a sí misma como oyente de eventos de ventana.

```
    public void windowOpened(WindowEvent e) {
        areaTexto.append(FIN + "Ventana abierta");
    }
    public void windowClosing(WindowEvent e) {
        areaTexto.append(FIN + "Ventana cerrandose");
    }
    public void windowClosed(WindowEvent e) {
        areaTexto.append(FIN + "Ventana cerrada");
    }
    public void windowIconified(WindowEvent e) {
        areaTexto.append(FIN + "Ventana iconizada");
    }
    public void windowDeiconified(WindowEvent e) {
        areaTexto.append(FIN + "Ventana desiconizada");
    }
    public void windowActivated(WindowEvent e) {
        areaTexto.append(FIN + "Ventana activada");
    }
    public void windowDeactivated(WindowEvent e) {
        areaTexto.append(FIN + "Ventana desactivada");
    }
    public static void main(String[] args) {
        VentanaOyente ventana = new VentanaOyente();
    }
}//VentanaOyente
```

Se proporcionan los métodos correspondientes a la interfaz `WindowListener`.

En los métodos de la interfaz `WindowListener` se añade una línea al contenido del área de texto con la indicación del evento producido.

Comentario: En este caso se están tratando los eventos de ventana que reflejan cualquier cambio de estado. Estos eventos son de bajo nivel. Siempre que sea posible, es preferible hacer el tratamiento de eventos de alto nivel o semánticos que hacerlo de sus eventos equivalentes de bajo nivel.

Aviso: Como se ha modificado el comportamiento del control de cierre de la ventana para que no haga nada y se pueda comprobar ese evento mediante su escritura en el área de texto, no se podrá cerrar la ventana de forma normal. Habrá que hacerlo cerrando la ventana de la consola Java o desde el administrador de tareas.

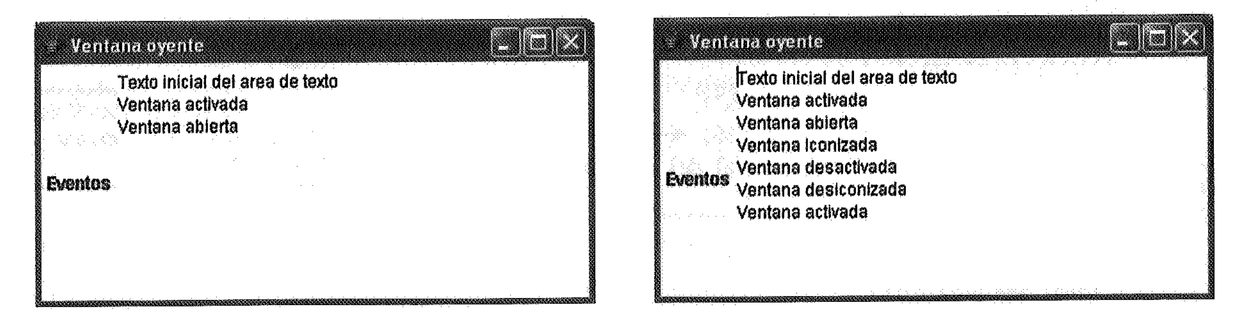


Figura 10.4. Ventana inicial de la aplicación y después de pasarla a icono y devolverla a su tamaño original.

Ejercicio 10.5:

Escriba una aplicación gráfica con una ventana que tenga un botón y un área de texto. La aplicación debe reflejar en el área de texto los principales eventos de ratón que se produzcan sobre dicha área por las interacciones del usuario. Haciendo clic en el botón se limpiará el contenido del área de texto.

Planteamiento: Primero se construye la ventana principal de la aplicación que contiene un panel en el que se incluyen el botón y el área de texto.

Para poder reflejar algunos de los eventos del ratón se crea una clase oyente de ratón que implemente la interfaz `MouseInputListener`. Esta interfaz incluye siete métodos y para no tener que implementarlos todos se hace que la clase oyente extienda de la clase adaptadora `MouseInputAdapter`. Esta clase adaptadora proporciona definiciones vacías de los siete métodos de modo que sólo se necesita implementar los métodos que resulten interesantes.

Solución:

```
import javax.swing.*;
import java.awt.event.*;
import javax.swing.event.*;
public class VentanaOyenteAdaptador extends JFrame {
    private JTextArea areaTexto;
    public VentanaOyenteAdaptador() {
        JButton boton = new JButton("Limpiar");
        areaTexto = new JTextArea(12, 30);
        JPanel panel = new JPanel();
        panel.add(boton);
        panel.add(areaTexto);
        add(panel);
        ActionListener oyenteBoton= new OyenteAccion();
        boton.addActionListener(oyenteBoton);
        MouseListener oyenteRaton= new OyenteRaton();
        areaTexto.addMouseListener(oyenteRaton);
    }
    class OyenteRaton extends MouseInputAdapter {
        public void mouseClicked(MouseEvent e) {
            areaTexto.append("Se ha hecho clic \n");
            areaTexto.append(" Posicion X: "+e.getX());
            areaTexto.append(" Posicion Y: "+e.getY());
        }
        public void mousePressed(MouseEvent e) {
            if ((e.getModifiers() & InputEvent.BUTTON3_MASK) != 0)
                areaTexto.append ("El botón pulsado es el de la derecha \n");
            areaTexto.append("Se ha pulsado el botón del ratón \n");
        }
        public void mouseReleased(MouseEvent e) {
            areaTexto.append("Se ha soltado el botón del ratón \n");
        }
        public void mouseEntered(MouseEvent e) {
            areaTexto.append("El ratón ha entrado en el componente \n");
        }
        public void mouseExited(MouseEvent e) {
            areaTexto.append("El ratón ha salido del componente \n");
        }
    }

    class OyenteAccion implements ActionListener{
        public void actionPerformed (ActionEvent evento){
            areaTexto.setText("");
```

También hay que importar los eventos de swing ya que es aquí donde se define `MouseInputAdapter`.

Se crea un oyente de ratón que se asocia al área de texto.

El oyente de ratón se crea extendiendo la clase adaptadora `MouseInputAdapter`.

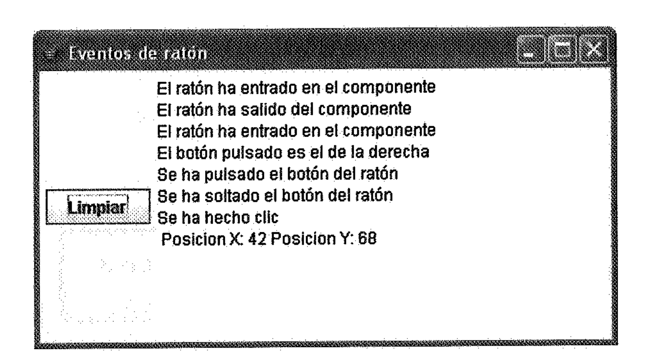
Oyente de acciones que limpia el contenido del área de texto.

```
        }
    }

    public static void main(String[] args) {
        VentanaOyenteAdaptador ventana = new VentanaOyenteAdaptador();
        ventana.setTitle("Eventos de ratón");
        ventana.pack();
        ventana.setVisible(true);
    }
}//VentanaOyenteAdaptador
```

En este caso las propiedades de la ventana de la aplicación se establecen en el programa principal.

Comentario: En este caso se están tratando los eventos de ratón a bajo nivel. El uso de la clase adaptadora ha evitado tener que proporcionar también la implementación de los métodos `mouseDragged()` y `mouseMoved()` ya que en este caso no se deseaba tener su funcionalidad.



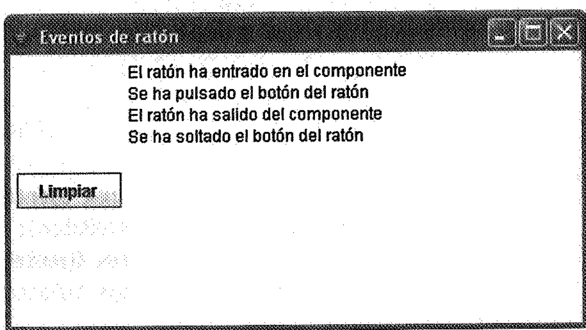


Figura 10.5. A la izquierda la ventana de la aplicación después de haber entrado, salido y vuelto a entrar en el área de texto donde se ha hecho clic con el botón derecho. A la derecha el ratón ha entrado en el área de texto, se ha pulsado el ratón y antes de soltarse se ha salido del componente de modo que no se ha producido el clic.

Ejercicio 10.6:

Escriba una aplicación gráfica que permita calcular el índice de masa corporal. Este índice se obtiene dividiendo el peso en kilos de una persona por el cuadrado de su altura en metros.

Planteamiento: Primero se construye la ventana principal de la aplicación que contiene un panel en el que se incluyen dos campos de texto con sus etiquetas identificativas para la obtención del peso y de la altura, un botón para realizar el cálculo y el campo de texto donde se muestra el resultado (este último campo no debe ser editable por el usuario). Los campos de texto que se necesitan para tomar datos o mostrar resultados, es decir, que serán posteriormente accedidos, se declaran como atributos de la clase. El resto de componentes gráficos se declaran como variables locales, o incluso anónimas.

Solución:

```
import javax.swing.*;
import java.awt.event.*;

public class IndiceMasaCorporal extends JFrame {
    private JTextField campoAltura;
    private JTextField campoPeso;
    private JTextField campoIMC;
```

```
    public IndiceMasaCorporal() {
        JLabel etiquetaAltura = new JLabel("Altura (metros)");
        JLabel etiquetaPeso = new JLabel("Peso (kg)");
        JLabel etiquetaIMC = new JLabel("Indice Masa Corporal");
        JButton calcular   = new JButton("Calcular IMC");
        campoAltura = new JTextField(6);
        campoPeso    = new JTextField(6);
        campoIMC     = new JTextField(6);
        campoIMC.setEditable(false);
        JPanel panel = new JPanel();

        panel.add(etiquetaAltura);
        panel.add(campoAltura);
        panel.add(etiquetaPeso);
        panel.add(campoPeso);
        panel.add(calcular);
        panel.add(etiquetaIMC);
        panel.add(campoIMC);
        add(panel);
        calcular.addActionListener(
            new ActionListener() {
                public void actionPerformed (ActionEvent evento){
                    Double peso = Double.parseDouble(campoPeso.getText());
                    Double altura = Double.parseDouble(campoAltura.getText());
                    Double imc = peso / (altura * altura);
                    String cadena = String.format("%6.2f", imc);
                    campoIMC.setText(cadena);
                }
            }
        );// addActionListener
    } //constructor
    public static void main(String[] args) {
        IndiceMasaCorporal ventana = new IndiceMasaCorporal();
        ventana.setTitle("Indice de Masa Corporal");
        ventana.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        ventana.setSize(400, 100);
        ventana.setVisible(true);
    }
}
```

Se crean los campos de altura y peso donde el usuario introducirá los datos.

Al botón de calcular se le asocia un oyente de acciones mediante una clase anónima.

Al resultado se le da formato con dos decimales y se muestra en el campoIMC.

Comentario: Normalmente en las aplicaciones gráficas sólo se declaran como atributos aquellos elementos gráficos a los que hay que acceder posteriormente (aunque en este caso no sería necesario). Respecto a la funcionalidad de la aplicación, en este caso se proporciona mediante un oyente de acciones asociado al botón de operación que se implementa como una clase anónima. Hay que tener cuidado con esta práctica ya que, aunque es cómodo debido a que no hay que crear nuevas clases, puede dificultar la lectura del código. Por tanto sólo se debe usar cuando el código del oyente sea suficientemente sencillo como para no interferir con la legibilidad del programa.

Aviso: No se ha introducido control de errores en la entrada proporcionada por el usuario, de modo que si el formato numérico no fuera correcto, se produciría una excepción y la correspondiente terminación anómala del programa. Si se introduce cero en la altura en el índice de masa corporal se muestra `Infinity` (infinito) ya que se ha realizado una división por cero.

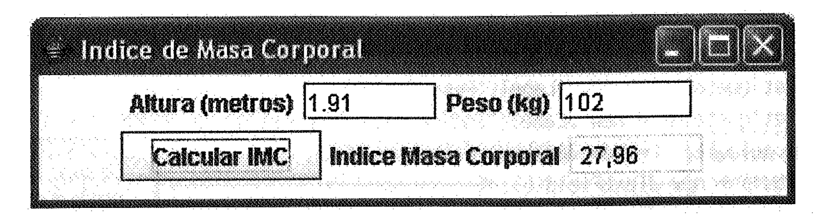


Figura 10.6. Captura de la ventana de la aplicación después de haber introducido datos y haber hecho clic en el botón Calcular IMC.

Ejercicio 10.7:

Escriba una aplicación gráfica que permita de forma sencilla realizar el cambio de pesetas a euros y viceversa. La tasa de cambio que se aplica es 1 euro igual a 166,386 pesetas. En todo momento el usuario debe estar informado de la conversión que se está realizando.

Planteamiento: En este caso se hace que la clase especialice a un panel (`JPanel`) en el que se incluyen los dos campos de texto con sus correspondientes etiquetas para obtener la cantidad a cambiar y mostrar el resultado. También se incluye un botón de operación para realizar el cambio y un botón conmutador que determinará la conversión que se está realizando en ese momento. Cuando se pulse el botón conmutador se cambiará la conversión a realizar y su texto identificativo para que el usuario conozca el sentido de dicha conversión, pesetas a euros o euros a pesetas.

En el programa principal se construye y se fijan las características de la ventana principal de la aplicación a la que se añade un objeto de la clase que especializa el panel.

Solución:

```
import javax.swing.*;
import java.awt.event.*;
public class CalculadoraEurosPesetas extends JPanel {
    private final double TASACAMBIO = 166.386;
    private double cambioEfectivo = TASACAMBIO;
    private JTextField campoCantidad;
    private JTextField campoResultado;

    public CalculadoraEurosPesetas() {
        add(new JLabel("Cantidad a convertir"));
        campoCantidad = new JTextField("0.0",6);
        add(campoCantidad);
        add(new JLabel("Resultado"));
        campoResultado = new JTextField("0.0",6);
        campoResultado.setEditable(false);
        add(campoResultado);
        JToggleButton moneda = new JToggleButton("Euros a Pesetas", false);
        add(moneda);
        moneda.addActionListener(new OyenteBotonConmutador());
        JButton cambiar = new JButton("Cambiar");
        add(cambiar);
        cambiar.addActionListener(new OyenteCambio());
    }

    class OyenteCambio implements ActionListener{
```

> En botón conmutador se fija que inicialmente no esté seleccionado. Luego se añade al panel y se le asocia un oyente de acciones.

> Oyente de acciones que obtiene la cantidad a cambiar, realiza la conversión que esté seleccionado y muestra el resultado (formateado con dos decimales).

```
        public void actionPerformed (ActionEvent evento){
            double dinero= Double.parseDouble(campoCantidad.getText());
            dinero = dinero * cambioEfectivo;
            String cadena = String.format("%6.2f", dinero);
            campoResultado.setText(cadena);
        }
    }// OyenteCambio

    class OyenteBotonConmutador implements ActionListener{
        public void actionPerformed (ActionEvent evento){
            JToggleButton boton = (JToggleButton) evento.getSource();
            if (boton.isSelected()){
                boton.setText("Pesetas a Euros");
                cambioEfectivo =  1 / TASACAMBIO;
            } else {
                boton.setText("Euros a Pesetas");
                cambioEfectivo = TASACAMBIO;
            }
        }
    }// OyenteBotonConmutador

    public static void main(String[] args) {
        JFrame ventana = new JFrame("Calculadora cambio moneda");
        CalculadoraEurosPesetas calculadora = new CalculadoraEurosPesetas();
        ventana.add(calculadora);
        ventana.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        ventana.setSize(400, 100);
        ventana.setVisible(true);
    }
}// CalculadoraEurosPesetas
```

Oyente del botón conmutador que cambia su etiqueta y establece el cambio efectivo.

Es necesario obtener el origen del evento ya que no está directamente accesible.

Se crea la ventana principal de la aplicación y se le proporciona el título en el constructor.

Se crea un objeto de tipo calculadora y se añade a la ventana principal.

Figura 10.7. Captura de la ventana de la aplicación primero en la conversión de euros a pesetas y luego en la de pesetas a euros. Obsérvese que el botón conmutador que muestra la conversión que se realiza cambia no sólo en su texto sino también su aspecto, ya que pasa a estar seleccionado.

Comentario: Como en este caso la clase principal no hereda de `JFrame` si no de `JPanel`, no produce una ventana principal en la que se muestra directamente la aplicación. Esta ventana se crea en el programa principal y luego se le añade un objeto del tipo calculadora.

Aviso: No se ha introducido control de errores en la entrada proporcionada por el usuario de modo que si el formato numérico no fuera correcto se produciría una excepción y la correspondiente terminación anómala del programa.

Ejercicio 10.8:

Mejore la aplicación gráfica que realiza el cambio de pesetas a euros y viceversa presentado en el ejercicio anterior. Además de las funcionalidades anteriores se debe permitir operar con los botones desde el teclado y hay que añadir un botón que permita borrar los campos de datos y resultado. También para darle mayor robustez se debe incluir un control de errores que avise mediante una ventana emergente si se ha introducido un número en formato erróneo evitando que la aplicación se detenga de forma anómala.

Planteamiento: En este caso se hace que la clase especialice a un panel (JPanel) en el que se incluyen los dos campos de texto con sus correspondientes etiquetas para obtener la cantidad a cambiar y mostrar el resultado. También se incluye un botón de operación para realizar el cambio y un botón conmutador que determinará la conversión que se está realizando en ese momento. Cuando se pulse el botón conmutador se cambiará la conversión a realizar y su texto identificativo para que el usuario conozca el sentido de dicha conversión (pesetas a euros o euros a pesetas). Para que se pueda operar con los botones desde el teclado se utilizan los mnemónicos. Si la letra indicada en el mnemónico aparece en el texto del botón, su primera ocurrencia aparecerá subrayada

En el programa principal se construye y se fijan las características de la ventana principal de la aplicación a la que se añade un objeto de la clase que especializa el panel.

Solución:

```
import javax.swing.*;
import java.awt.event.*;
import javax.swing.event.*;

public class CalculadoraEuroMejorada extends JPanel {
    private final double TASACAMBIO = 166.386;
    private double cambioEfectivo = TASACAMBIO;
    private JTextField campoCantidad;
    private JTextField campoResultado;

    public CalculadoraEuroMejorada() {
        add(new JLabel("Cantidad a convertir"));
        campoCantidad = new JTextField("0.0",6);
        add(campoCantidad);
        add(new JLabel("Resultado"));
        campoResultado = new JTextField("0.0", 6);
        campoResultado.setEditable(false);
        add(campoResultado);
        JToggleButton moneda = new JToggleButton("Euros a Pesetas", false);
        moneda.setMnemonic(KeyEvent.VK_E);
        add(moneda);
        moneda.addChangeListener(new OyenteBotonConmutador());
        JButton cambiar = new JButton("Cambiar");
        cambiar.setMnemonic(KeyEvent.VK_C);
        add(cambiar);
        cambiar.addActionListener(new OyenteCambio());
        ImageIcon icono = new ImageIcon("cross.gif");
```

Pulsando ALT+e se activa este botón. Si la letra e aparece en el texto del botón su primera ocurrencia aparecerá subrayada.

Se crea un icono a partir de una imagen gráfica en formato gif.

```
        JButton borrar = new JButton("Borrar", icono);
        borrar.setMnemonic(KeyEvent.VK_B);
        add(borrar);
        borrar.addActionListener(
            new ActionListener(){
                public void actionPerformed(ActionEvent ev) {
                    borrarDatos();
                }
            }

         );
    }
    private void borrarDatos(){
        campoCantidad.setText("0.0");
        campoResultado.setText("0.0");
    }

    class OyenteCambio implements ActionListener{
        public void actionPerformed (ActionEvent evento){
            double dinero= 0.0;
            try{
                dinero = Double.parseDouble(campoCantidad.getText());
            } catch (NumberFormatException excepcion){
                JOptionPane.showMessageDialog(CalculadoraEuroMejorada.this,
                    "Solo se pueden introducir dígitos y el punto decimal",
                    "Error en el formato numérico",
                    JOptionPane.ERROR_MESSAGE);
                borrarDatos();
            }
            dinero = dinero * cambioEfectivo;
            String cadena = String.format("%6.2f", dinero);
            campoResultado.setText(cadena);
        }
    }// OyenteCambio

    class OyenteBotonConmutador implements ChangeListener{
        public void stateChanged(ChangeEvent evento){
            JToggleButton boton = (JToggleButton) evento.getSource();
            if (boton.isSelected()){
                boton.setText("Pesetas a Euros");
                cambioEfectivo = 1 / TASACAMBIO;
            } else {
                boton.setText("Euros a Pesetas");
                cambioEfectivo = TASACAMBIO;
            }
        }
    }// OyenteBotonConmutador

    public static void main(String[] args) {
        JFrame ventana = new JFrame("Calculadora cambio moneda");
        CalculadoraEuroMejorada calculadora = new CalculadoraEuroMejorada();
```

Se crea un botón que además de un texto tiene un icono.

Se añade un oyente anónimo que invoca al método privado de borrado de datos.

Método privado de la clase que permite mejorar la estructuración de código.

Se protege la lectura de datos para evitar que un error en el formato numérico provoque la terminación anómala de la ejecución.

Se crea una ventana emergente con el mensaje de error y un icono que muestra dicho error.

```
        ventana.add(calculadora);
        ventana.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        ventana.setSize(400, 130);
        ventana.setVisible(true);
    }
}// CalculadoraEuroMejorada
```

Comentario: La imagen gráfica "cross.gif" es la que se incluye en las distribuciones estándar de JDK de Java. Normalmente se puede encontrar en la ruta demo/applets/TicTacToe/images a partir del directorio de instalación del JDK.

En este caso se ha modificado el oyente del botón conmutador a un oyente de cambios. Su comportamiento es similar y también es un evento semántico o de alto nivel.

Cuando se crea el cuadro de diálogo emergente como primer argumento se debe proporcionar el componente gráfico, normalmente una ventana, del que depende dicho cuadro emergente. Como el oyente es una clase interna al panel se podría haber tratado de utilizar this pero en ese caso representa la instancia del oyente, que no es un componente gráfico, y por tanto hay que usar la expresión CalculadoraEuroMejorada.this mediante la cual se accede a la instancia actual de la clase contenedora del oyente. También podría haberse utilizado null como argumento, pero en este caso no se tiene referencia sobre dónde posicionar el cuadro de diálogo emergente.

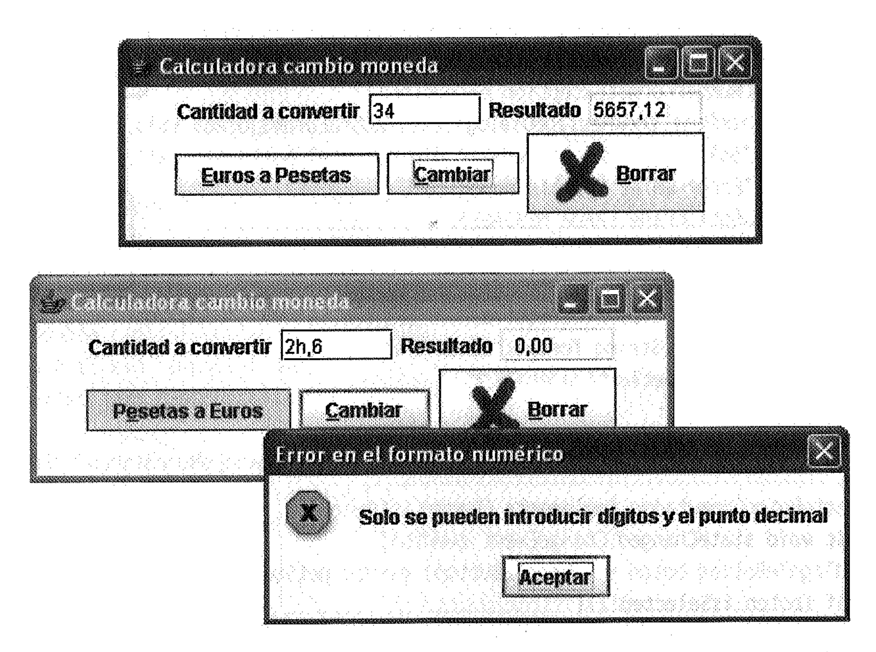


Figura 10.8. Captura de la ventana de la aplicación que ya incorpora el botón Borrar con el incono de la cruz roja y donde en el texto de los botones aparecen subrayadas las letras elegidas como mnemónicos, en el botón de selección del tipo de cambio se sigue subrayando la primera letra e incluso cuando se cambia de etiqueta. En la segunda captura se muestra el cuadro de diálogo emergente donde se indica que se ha producido un error en el formato numérico.

Ejercicio 10.9:

Escriba una aplicación gráfica que cree un tablero de tamaño de ocho por ocho con cuadros blancos y negros similar al utilizado en el juego de las damas o del ajedrez. La aplicación debe detectar la pulsación sobre el tablero e informar sobre el color del cuadro pulsado.

Planteamiento: En este caso se hace que la clase especialice a un panel (JPanel) en el que se incluye un array de botones que representarán los cuadros. Para que todos los cuadros tengan el mismo tamaño se elige que su tamaño preferido sea cuadrado, mediante setPreferredSize(). Para representar los cuadros negros se establece su color de fondo a dicho color (setBackground(Color.BLACK)). Por defecto, el gestor de disposición o administrador de diseño, en inglés LayoutManager, del panel es el FlowLayout que organiza los componentes en filas según el orden en el que se han añadido al panel. Como interesa organizar los cuadros en forma de tablero se cambia el gestor de disposición por defecto por el gestor GridLayout(filas, columnas) que organiza los componentes en una matriz de tamaño dado por las filas y las columnas.

En el programa principal se construye y se fijan las características de la ventana principal de la aplicación a la que se añade un objeto de la clase que especializa el panel.

Solución:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class Tablero extends JPanel {
    public Tablero(int tamaño) {
        JButton[][] botones;
        botones = new JButton[tamaño][tamaño];
        OyenteAcciones oyente = new OyenteAcciones(this);
        for (int i = 0; i< botones.length; i++)
            for (int j = 0; j<botones[i].length; j++){
                botones[i][j]= new JButton();
                botones[i][j].setPreferredSize(new Dimension(50, 50));
                if (((i+j+1) % 2) == 0){
                    botones[i][j].setBackground(Color.BLACK);
                }
                botones[i][j].addActionListener(oyente);
                add(botones[i][j]);
        }
        setLayout(new GridLayout(tamaño, tamaño));
    }

    class OyenteAcciones implements ActionListener{
        private JPanel panel;
        public OyenteAcciones(JPanel panel){
            this.panel = panel;
        }
        public void actionPerformed (ActionEvent evento){
            JButton boton = (JButton)evento.getSource();
            String color = "blanco";
            if (boton.getBackground()== Color.BLACK)
                color = "negro";
            JOptionPane.showMessageDialog(panel,
                    "Se ha pulsado un cuadro de color " + color,
                    "Cuadro pulsado",
                    JOptionPane.INFORMATION_MESSAGE);

        }
    } //OyenteAcciones
```

Se crea un oyente de acciones y se le pasa el panel como argumento.

Se crean los botones y se establece su tamaño preferido a 50 por 50 píxeles.

Se establece el gestor de disposición GridLayout para el panel.

Se obtiene el color del botón pulsado.

Se muestra un cuadro de diálogo informativo.

El icono que se utilizará en el cuadro emergente es la letra i de información.

```
    public static void main(String[] args) {
        JFrame ventana = new JFrame("Tablero");
        Tablero tablero = new Tablero(8);
        ventana.add(tablero);
        ventana.pack();
        ventana.setVisible(true);
    }
}
```

Comentario: El oyente de acciones tiene un constructor parametrizado que recibe como parámetro el panel en el que está incluido el botón al que se escucha.

En el cálculo del tamaño de la ventana mediante `pack()` se tiene en cuenta el tamaño preferido de cada uno de los componentes que contiene

Si hubiera interesado además del color cuáles son las coordenadas del cuadro pulsado se podría haber creado una clase que especializara un `JButton` y tuviera dos atributos privados `coordenadaX` y `coordenadaY`.

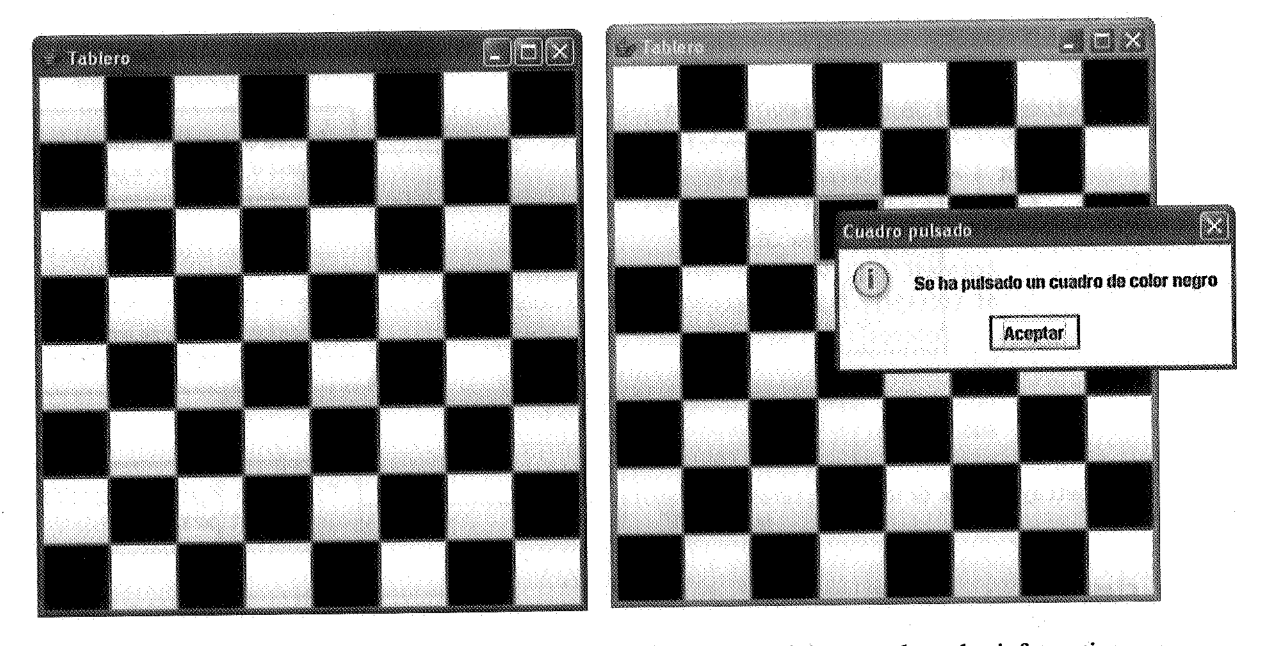


Figura 10.9. Captura de la ventana de la aplicación con el tablero inicial y con el cuadro informativo que emerge después de haber pulsado un cuadro de color negro.

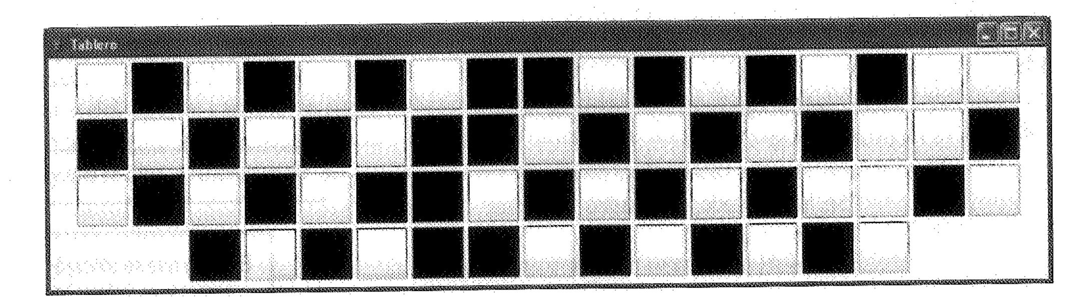


Figura 10.10. Si no se hubiera asignado expresamente el gestor de disposición por defecto en un panel, es el FlowLayout quien lo haría y se hubiera obtenido una presentación similar a la de esta captura.

Ejercicio 10.10:

Escriba una aplicación gráfica que permita comprobar la importancia de la orientación del componente y de la alineación en el gestor de disposición FlowLayout. El usuario debe poder cambiar mediante interacción sencilla la orientación y la alineación en la interfaz gráfica y dichos cambios se deben reflejar inmediatamente.

Planteamiento: Crear una ventana con tres paneles. El panel central permitirá comprobar los efectos de la orientación y la alineación con el gestor de disposición `FlowLayout`, que es el que tienen asignado por defecto los paneles. Para ello se rellenará con botones de operación, que se etiquetarán un texto identificativo del orden en el que se añaden al panel y se dejará suficiente espacio libre de modo que se observe bien los efectos de la orientación y la alineación. En el panel superior se permitirá que el usuario seleccione la orientación que se desea aplicar al panel central de botones. Para ello se incluirán dos botones de radio o selección de modo que el usuario sólo pueda seleccionar uno de ellos cada vez. En el panel inferior también mediante botones de radio se permitirá que el usuario seleccione alguna de las cinco posibles alineaciones. Como las ventanas de tipo `JFrame` tienen asignado por defecto el gestor de disposición `BorderLayout`, se usan 3 de sus 5 regiones (principio de página, centro y final de página) para colocar dichos paneles en la ventana de la aplicación.

El código se estructurará en métodos privados siempre que sea posible para simplificar la legibilidad y mantenibilidad del código evitando que el constructor crezca mucho de tamaño.

Solución:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class EjemploLayouts extends JFrame {
    JPanel panelBotones;
    public EjemploLayouts() {
        panelBotones = crearPanelBotones();
        add(panelBotones, BorderLayout.CENTER);
        JPanel panelOrientacion = crearPanelOrientacion();
        add(panelOrientacion, BorderLayout.PAGE_START);
        JPanel panelAlineacion = crearPanelAlineacion();
        add(panelAlineacion, BorderLayout.PAGE_END);
        setTitle("FlowLayout: orientación y alineación");
        setSize(500, 200);
        setVisible(true);
    }

    private JPanel crearPanelBotones(){
        JPanel panelBotones = new JPanel();
        JButton botones[] = new JButton[8];
        for (int i=0; i<botones.length; i++){
            botones[i] = new JButton("Boton "+ i);
            panelBotones.add(botones[i]);
        }
        return panelBotones;
    } //crearPanelBotones

    private JPanel crearPanelOrientacion(){
      JPanel panel = new JPanel();
      OyenteOrientacion oyente = new OyenteOrientacion();
```

Panel de botones de radio para seleccionar la orientación que se añade en la parte superior de la ventana.

Panel de botones de radio para seleccionar la alineación que se añade en la parte inferior de la ventana.

```
        ButtonGroup grupoOrientacion = new ButtonGroup();
        panel.add(new JLabel("Orientacion: "));
        panel.add(crearBotonRadio(grupoOrientacion,"LEFT_TO_RIGHT", oyente));
        panel.add(crearBotonRadio(grupoOrientacion,"RIGHT_TO_LEFT", oyente));
        return panel;
    }//crearPanelOrientacion

    private JPanel crearPanelAlineacion(){
        JPanel panel = new JPanel();
        OyenteAlineacion oyente = new OyenteAlineacion();
        ButtonGroup grupoAlineacion = new ButtonGroup();
        panel.add(new JLabel("Alineación: "));
        panel.add(crearBotonRadio(grupoAlineacion, "LEFT", oyente));
        panel.add(crearBotonRadio(grupoAlineacion, "CENTER", oyente));
        panel.add(crearBotonRadio(grupoAlineacion, "RIGHT", oyente));
        panel.add(crearBotonRadio(grupoAlineacion, "LEADING", oyente));
        panel.add(crearBotonRadio(grupoAlineacion, "TRAILING", oyente));
        return panel;
    }//crearPanelAlineacion

    private JRadioButton crearBotonRadio(ButtonGroup grupo,
                String etiqueta, ActionListener oyente){
        JRadioButton boton = new JRadioButton(etiqueta);
        boton.addActionListener(oyente);
        grupo.add(boton);
        return boton;
    }//crearBotonRadio

    class OyenteOrientacion implements ActionListener{
        public void actionPerformed (ActionEvent evento){
            JRadioButton boton = (JRadioButton)evento.getSource();
            if ("LEFT_TO_RIGHT".equals(boton.getActionCommand())){
                panelBotones.setComponentOrientation(
                        ComponentOrientation.LEFT_TO_RIGHT);
            } else {
                panelBotones.setComponentOrientation(
                        ComponentOrientation.RIGHT_TO_LEFT);
            }
            panelBotones.doLayout();
        }
    }//OyenteOrientacion

    class OyenteAlineacion implements ActionListener{
        public void actionPerformed (ActionEvent evento){
            JRadioButton boton = (JRadioButton)evento.getSource();
            FlowLayout layout = (FlowLayout)panelBotones.getLayout();
            if ("LEFT".equals(boton.getActionCommand())){
                layout.setAlignment(FlowLayout.LEFT);
            } else if ("CENTER".equals(boton.getActionCommand())){
                layout.setAlignment(FlowLayout.CENTER);
            }else if ("RIGHT".equals(boton.getActionCommand())){
```

Para que los botones de radio sean excluyentes entre sí deben estar asociados al mismo grupo de botones.

El método getActionCommand() tiene un comportamiento similar a getText().

Se cambia la orientación del panel de botones.

```
                layout.setAlignment(FlowLayout.RIGHT);
            }else if ("LEADING".equals(boton.getActionCommand())){
                layout.setAlignment(FlowLayout.LEADING);
            }else{
                layout.setAlignment(FlowLayout.TRAILING);
            }
            panelBotones.doLayout();
        }
    }//OyenteAlineacion

    public static void main(String[] args) {
        EjemploLayouts ventana = new EjemploLayouts();
    }
}
```

Comentario: Observe cómo los métodos privados permiten estructurar el código al igual que se hace con las clases no gráficas. Esto permite evitar constructores muy largos que dificultan la legibilidad.

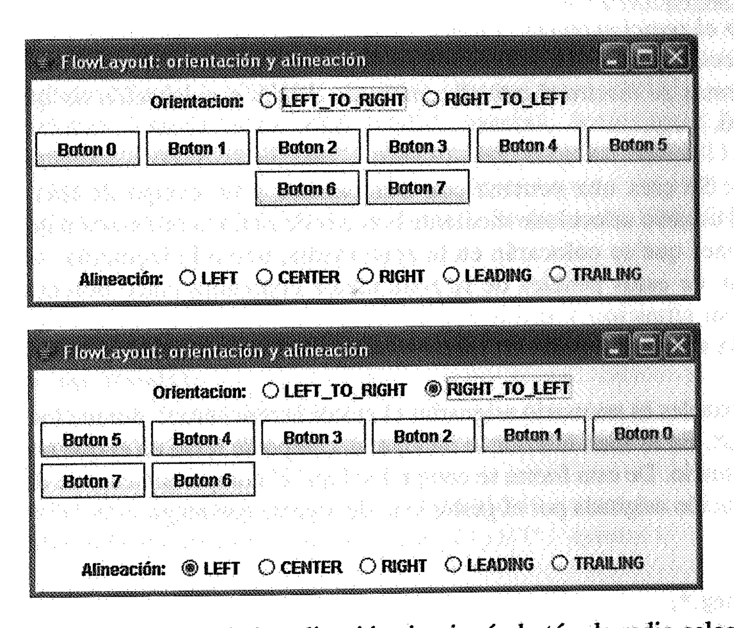


Figura 10.11. Captura inicial de la ventana de la aplicación sin ningún botón de radio seleccionado, por defecto se presenta en orientación izquierda derecha y alineación centrada. Como se puede ver en la captura de abajo, si se cambia la orientación los botones se añaden de derecha a izquierda y si la alineación es izquierda en la siguiente línea se deja libre el espacio sobrante por la derecha.

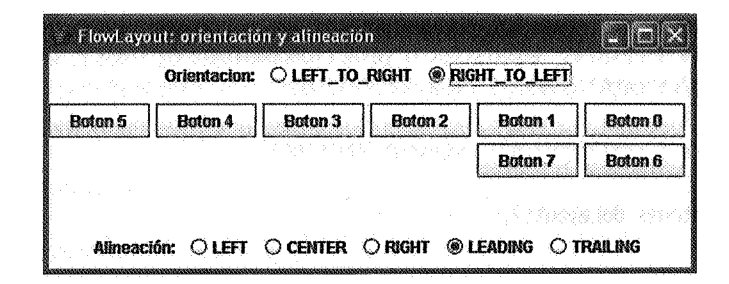


Figura 10.12. La alineación `LEADING` es relativa a la orientación del componente. En la primera captura, como la orientación es izquierda a derecha, su comportamiento es similar a la alineación `LEFT`. En la segunda captura, como la orientación es de derecha a izquierda, su comportamiento es similar a la alineación `RIGHT`.

Ejercicio 10.11:

Escriba una aplicación gráfica que permita comprobar el funcionamiento del gestor de disposición BorderLayout. Se debe poder comprobar la importancia de la orientación del componente y como ésta afecta a la colocación de los componentes según la especificación de colocación utilizada (la basada en regiones "geográficas" o el posicionamiento usando términos de líneas y páginas). El usuario debe poder cambiar mediante interacción sencilla la orientación en la interfaz gráfica y dichos cambios se deben reflejar inmediatamente. Además se mostrará que el cambio de diseño o colocación no afecta al comportamiento de dichos elementos.

Planteamiento: Se crea una ventana con tres paneles y un campo de texto. En el panel superior se permitirá que el usuario seleccione mediante botones de radio la orientación que se desea aplicar a los dos paneles de botones que se colocarán en la zona media, uno a la izquierda –`WEST`– y otro a la derecha –`EAST`–. Cada uno de estos paneles de la zona media contendrá cinco botones etiquetados con un texto descriptivo de su situación y usará una de las posibles especificaciones de colocación en el gestor `BorderLayout`, la absoluta basada en regiones o la relativa utilizando el orden aplicado a la escritura de texto.

A estos dos paneles es necesario asignarles el gestor `BorderLayout` porque los paneles tiene por defecto el gestor `FlowLayout`. En la zona inferior se incluye un campo de texto en el que se mostrará qué botón ha sido pulsado por el usuario. De esta forma se comprobará que el comportamiento de dicho botón no depende de la colocación o posición asignada por el gestor sino del oyente que tenga.

Solución:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class PruebaBorderLayout extends JFrame {
    JPanel panelBotonesRegiones;
    JPanel panelBotonesRelativos;
    JTextField campoTexto;

    public PruebaBorderLayout(){
      add(crearPanelOrientacion(), BorderLayout.NORTH);
      panelBotonesRegiones = crearPanelRegiones();
      add(panelBotonesRegiones, BorderLayout.EAST);
      panelBotonesRelativos = crearPanelPosicionRelativa();
      add(panelBotonesRelativos, BorderLayout.WEST);
```

```
    campoTexto = new JTextField();
    add(campoTexto, BorderLayout.SOUTH);
}

private JPanel crearPanelRegiones(){ <-- Se crea el panel de botones con especificación de regiones absolutas.
    JPanel panel = new JPanel();
    panel.setLayout(new BorderLayout());
    OyenteBoton oyente = new OyenteBoton();
    panel.add(crearBoton("Botón 1 (CENTER)", oyente), BorderLayout.CENTER);
    panel.add(crearBoton("Segundo Botón (NORTH)", oyente), BorderLayout.NORTH);
    panel.add(crearBoton("3 Botón (SOUTH)", oyente), BorderLayout.SOUTH);
    panel.add(crearBoton("Bot. 4 (EAST)", oyente), BorderLayout.EAST);
    panel.add(crearBoton("Botón5 (WEST)", oyente), BorderLayout.WEST);
    return panel;
}

private JPanel crearPanelPosicionRelativa(){ <-- Se crea el panel de botones con especificación relativa a como se escribe el texto en una página.
    JPanel panel = new JPanel();
    panel.setLayout(new BorderLayout());
    OyenteBoton oyente = new OyenteBoton();
    panel.add(crearBoton("Botón 1 (CENTER)", oyente), BorderLayout.CENTER);
    panel.add(crearBoton("Segundo Botón (PAGE_START)", oyente), BorderLayout.PAGE_START);
    panel.add(crearBoton("3 Botón (PAGE_END)", oyente), BorderLayout.PAGE_END);
    panel.add(crearBoton("Bot. 4 (LINE_END)", oyente), BorderLayout.LINE_END);
    panel.add(crearBoton("Botón5 (LINE_START)", oyente), BorderLayout.LINE_START);
    return panel;
}

private JPanel crearPanelOrientacion(){
    JPanel panel = new JPanel();
    OyenteOrientacion oyente = new OyenteOrientacion();
    ButtonGroup grupoOrientacion = new ButtonGroup();
    panel.add(new JLabel("Orientacion: "));
    panel.add(crearBotonRadio(grupoOrientacion,"LEFT_TO_RIGHT", oyente));
    panel.add(crearBotonRadio(grupoOrientacion,"RIGHT_TO_LEFT", oyente));
    return panel;
}

private JRadioButton crearBotonRadio(ButtonGroup grupo, String etiqueta,
                          ActionListener oyente){
    JRadioButton boton= new JRadioButton(etiqueta);
    boton.addActionListener(oyente);
    grupo.add(boton);
    return boton;
}

private JButton crearBoton(String etiqueta, ActionListener oyente){
    JButton boton = new JButton(etiqueta);
    boton.addActionListener(oyente);
    return boton;
}
```

```
class OyenteBoton implements ActionListener{
   public void actionPerformed (ActionEvent evento){
      JButton boton = (JButton)evento.getSource();
      campoTexto.setText("Se ha pulsado el boton: "+ boton.getText());
   }
}

class OyenteOrientacion implements ActionListener{
   public void actionPerformed (ActionEvent evento){
      JRadioButton boton = (JRadioButton)evento.getSource();
      if ("LEFT_TO_RIGHT".equals(boton.getActionCommand())){
         panelBotonesRegiones.setComponentOrientation(
                          ComponentOrientation.LEFT_TO_RIGHT);
         panelBotonesRelativos.setComponentOrientation(
                          ComponentOrientation.LEFT_TO_RIGHT);
      }else{
         panelBotonesRegiones.setComponentOrientation(
                          ComponentOrientation.RIGHT_TO_LEFT);
         panelBotonesRelativos.setComponentOrientation(
                          ComponentOrientation.RIGHT_TO_LEFT);
      }
      panelBotonesRegiones.doLayout();
      panelBotonesRelativos.doLayout();
   }
}//OyenteOrientacion

public static void main(String args[]) {
   PruebaBorderLayout ventana = new PruebaBorderLayout();
   ventana.setTitle("Administrador BorderLayout");
   ventana.setSize(900,250);
   ventana.setVisible(true);
}
}
```

Cuando se cambia la orientación se aplica la nueva a los dos paneles de botones y se rehace su presentación.

Comentario: Fíjese en las figuras con las capturas de la aplicación que el gestor de diseño BorderLayout no asigna por igual el espacio para las cinco zonas. Si se redimensiona la ventana, la parte central tendrá asignado más espacio.

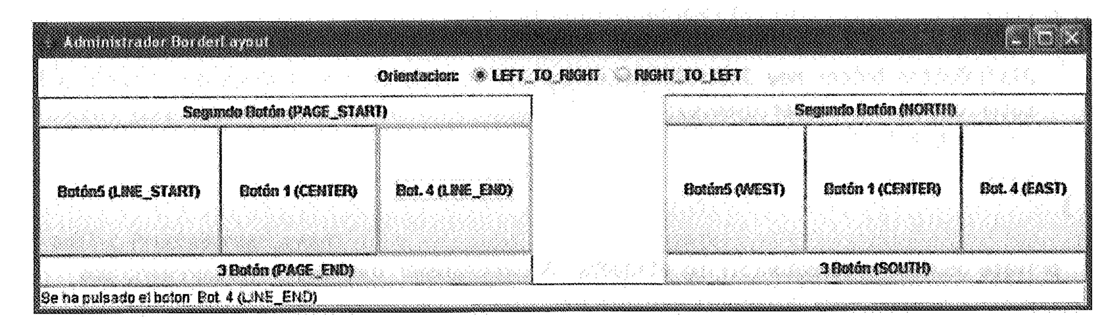


Figura 10.13. Captura de la ventana de la aplicación después de haber seleccionado orientación izquierda derecha (que es la que se tiene por defecto) y haber pulsado el botón 4 en el panel de posicionamiento relativo.

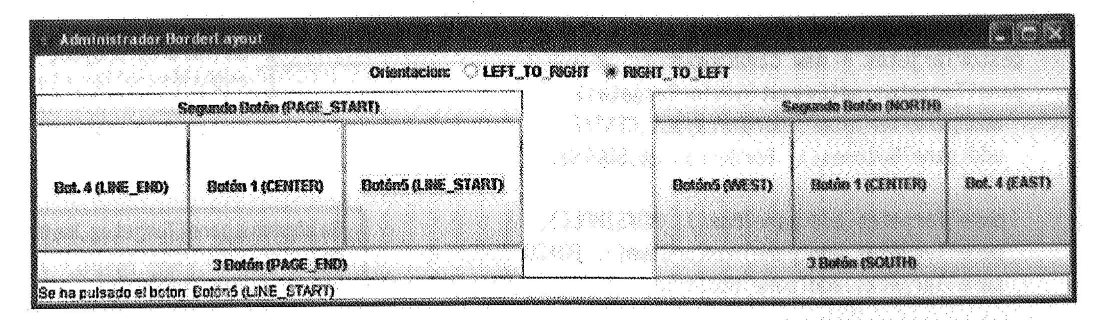


Figura 10.14. Captura de la ventana de la aplicación después de haber cambiado la orientación a derecha izquierda. En este caso los botones 4 y 5 en el posicionamiento relativo intercambian su posición. Los botones del panel derecho que tienen posicionamiento por regiones no modifican su colocación.

Ejercicio 10.12:

Escriba una aplicación gráfica que utilizando el gestor de disposición CardLayout permita comprobar algunas características del funcionamiento del gestor de disposición BoxLayout. Se deberá mostrar cómo coloca en la pantalla de la aplicación cinco botones de distinto tamaño tanto en sentido vertical como en sentido horizontal y cómo se puede distribuir el espacio sobrante entre los botones cuando se redimensiona la ventana. Además se debe mostrar cómo se pueden separar dos grupos de botones.

Planteamiento: Se crea una ventana con dos paneles y una etiqueta. En la parte superior se coloca la etiqueta que identifica el ejemplo que se está mostrando en ese momento. En la parte central se coloca un panel con un gestor de disposición `CardLayout` que a su vez contiene tres paneles con cinco botones cada uno que son los tres ejemplos de uso de `BoxLayout`. Por el funcionamiento del `CardLayout`, en cada momento sólo puede estar visible uno de ellos. En la parte inferior de la ventana se coloca un panel con tres botones de operación que permite el cambio entre los tres paneles de ejemplo del panel central. Al pulsar cada botón se pasa a visualizar el panel correspondiente en el área central y se modifica la etiqueta acordemente para informar al usuario.

Por defecto, el gestor de disposición `BoxLayout` coloca todos los componentes agrupados desde un extremo del contenedor en función de la orientación de dicho contenedor y deja el espacio sobrante al final. Para redistribuir este espacio sobrante si existe se puede incluir elementos invisibles de relleno, que pueden ser rígidos como las áreas rígidas o los struts o bien flexibles como los pegamentos (glue). Si se añade pegamento entre todos los botones, entonces éstos se distribuyen de forma regular entre el espacio disponible.

Solución:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class BoxCardLayout extends JFrame {
    final static String BOXSIMPLE = "BoxLayout simple";
    final static String BOXCONGLUE = "BoxLayout con glue";
    final static String BOXCONSTRUT = "BoxLayout con strut";
    private CardLayout gestorTarjetas;
    private JPanel panelTarjetas;
    private JLabel etiqueta;

    BoxCardLayout(){
        etiqueta = new JLabel(BOXSIMPLE);
        add(etiqueta, BorderLayout.NORTH);
```

Cadenas que identifican a cada uno de los paneles del `CardLayout`.

```
        panelTarjetas = new JPanel();
        gestorTarjetas = new CardLayout();
        panelTarjetas.setLayout(gestorTarjetas);
        add(panelTarjetas, BorderLayout.CENTER);
        add(panelBotones(), BorderLayout.SOUTH);

        panelTarjetas.add(panelBox(), BOXSIMPLE);
        panelTarjetas.add(panelBoxConGlue(), BOXCONGLUE);
        panelTarjetas.add(panelBoxStrut(),BOXCONSTRUT);

        setTitle("Pruebas de BoxLayout y CardLayout");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        pack();
        setVisible(true);
    }

    private JPanel panelBotones(){
        JPanel panel = new JPanel();
        OyenteBoton oyente = new OyenteBoton();
        JButton bs = new JButton(BOXSIMPLE);
        bs.addActionListener(oyente);
        panel.add(bs);
        JButton bg = new JButton(BOXCONGLUE);
        bg.addActionListener(oyente);
        panel.add(bg);
        JButton bstrut = new JButton(BOXCONSTRUT);
        bstrut.addActionListener(oyente);
        panel.add(bstrut);
        return panel;
    }

    private JPanel panelBoxConGlue(){
        JPanel panel = new JPanel();
        panel.setLayout( new BoxLayout(panel, BoxLayout.PAGE_AXIS));
        panel.add(new JButton("Botón 1"));
        panel.add(Box.createGlue());
        panel.add(new JButton("Segundo Botón"));
        panel.add(Box.createGlue());
        panel.add(new JButton("3 Botón"));
        panel.add(Box.createGlue());
        panel.add(new JButton("Bot. 4"));
        panel.add(Box.createGlue());
        panel.add(new JButton("Botón5"));
        return panel;
    }

    private JPanel panelBox(){
        JPanel panel = new JPanel();
        panel.setLayout( new BoxLayout(panel, BoxLayout.LINE_AXIS));
        panel.add(new JButton("Botón 1"));
        panel.add(new JButton("Segundo Botón"));
```

Se crea el CardLayout y se le asigna al panelTarjetas.

Se añaden a panelTarjetas los tres paneles de botones organizados con BoxLayout cada uno con su texto identificativo.

Los botones se organizan verticalmente.

Entre los botones se añade pegamento.

Los botones se organizan horizontalmente.

```
            panel.add(new JButton("3 Botón"));
            panel.add(new JButton("Bot. 4"));
            panel.add(new JButton("Botón5"));
          return panel;
      }

      private JPanel panelBoxStrut(){
          JPanel panel = new JPanel();
            panel.setLayout( new BoxLayout(panel, BoxLayout.LINE_AXIS));
          panel.add(new JButton("Botón 1"));
            panel.add(new JButton("Segundo Botón"));
          panel.add(new JButton("3 Botón"));
            panel.add(Box.createHorizontalStrut(10)); <-- Entre los botones 3 y 4 se introduce como elemento de relleno un strut que no cambia de tamaño.
          panel.add(new JButton("Bot. 4"));
          panel.add(new JButton("Botón5"));
          return panel;
      }

      class OyenteBoton implements ActionListener{ <-- Coloca como panel visible del panel central el panel identificado por el texto del botón.
          public void actionPerformed (ActionEvent evento){
             JButton boton = (JButton)evento.getSource();
             etiqueta.setText(boton.getText());
             gestorTarjetas.show(panelTarjetas, boton.getText());
          }
      }
      public static void main(String args[]) {
          BoxCardLayout ventana = new BoxCardLayout();
      }
  }
```

Comentario: Se podría haber colocado un único botón y haber ido recorriendo de forma circular todos los paneles gestionados por el `CardLayout` simplemente haciendo que en dicho oyente se ejecutara la instrucción `gestorTarjetas.next()`. Funciona de forma circular y desde la última tarjeta (en este caso panel) se pasa a visualizar la primera.

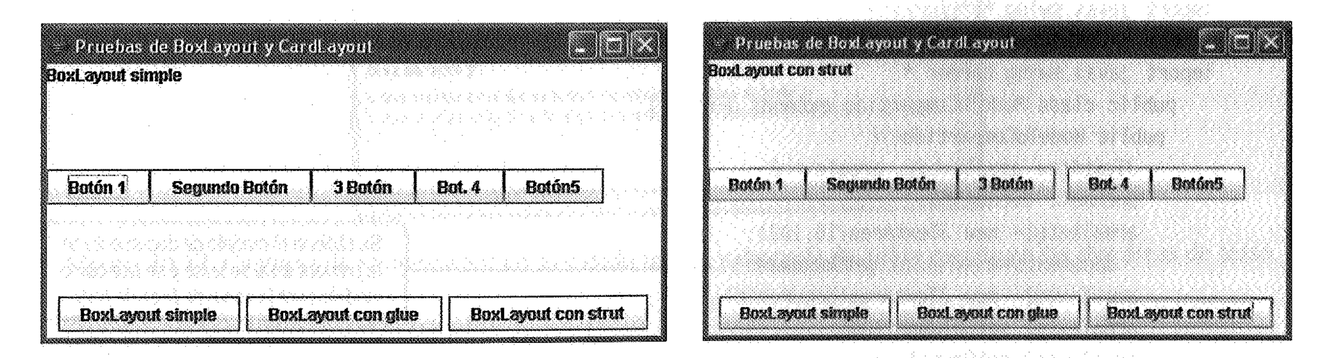


Figura 10.15. Capturas de la aplicación mostrando dos de los tres paneles de botones gestionados por el CardLayout. Obsérvese que los botones se han organizado horizontalmente. En la primera todo el espacio libre queda a la derecha de los botones mientras que en la segunda existe una separación entre los botones 3 y 4 debido a que se introdujo un strut.

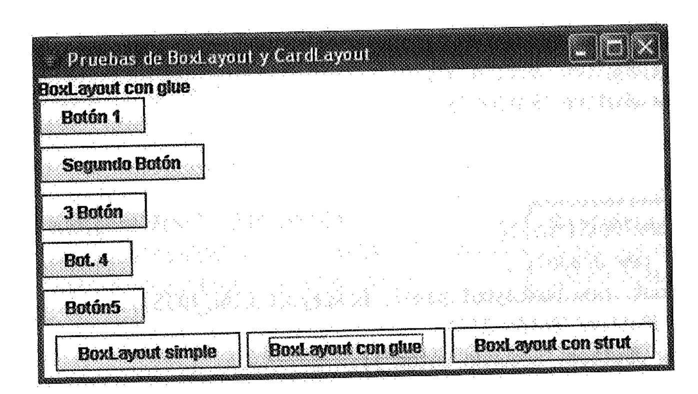


Figura 10.16. Captura de la aplicación mostrando el tercer panel de botones gestionado por el CardLayout. Obsérvese que los botones se han organizado verticalmente y en este caso el espacio libre está repartido entre los botones debido a que entre ellos se introdujo pegamento (glue).

Ejercicio 10.13:

Escriba una aplicación gráfica que presente dos áreas de texto editables y cuyo comportamiento sea que lo que se escriba en cada una de ellas se refleje automáticamente en la otra.

Planteamiento: Se crea una ventana con dos áreas de texto (JTextArea) editables. Se podría tratar de lograr el comportamiento solicitado mediante oyentes de acciones que tomen el texto escrito en una ventana y se lo añadan al contenido de la otra ventana. Sin embargo hay otra solución mas directa que es aprovechar la arquitectura modelo-vista-controlador (en concreto modelo-delegado) que incorpora Swing y usar el modelo de documento que tienen las áreas de texto. Este modelo refleja el contenido del área de texto y está siempre sincronizado con la vista gráfica de dicha área de texto. Por tanto si se comparte el mismo modelo entre las dos áreas de texto ya está resuelto el problema.

Para mejorar la legibilidad y la presentación gráfica se añade un borde con título a las áreas de texto. Por si el texto sobrepasa el tamaño visible asignado al área de texto se presentan dentro de paneles con desplazamiento de modo que las barras de desplazamiento permitan visualizar dicho texto.

Solución:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.text.*;
import javax.swing.border.*;
  public class ModeloCompartido extends JFrame {
   public ModeloCompartido(){
     JTextArea areaTexto1, areaTexto2;
     Document documento;
     areaTexto1= new JTextArea(10,100);
       documento=areaTexto1.getDocument();
     areaTexto2= new JTextArea(documento);
     areaTexto2.setColumns(100);
     areaTexto2.setRows(10);
     TitledBorder titulo1;
       titulo1 = BorderFactory.createTitledBorder("Area de texto 1");
       areaTexto1.setBorder(titulo1);
     TitledBorder titulo2;
```

Se obtiene el modelo de documento de la primera área de texto y se usa como modelo para la segunda área de texto.

Se crea un borde con título y se le asigna a cada área de texto para mejorar la presentación gráfica.

```
        titulo2 = BorderFactory.createTitledBorder("Area de texto 2");
        areaTexto2.setBorder(titulo2);
        JPanel panel = new JPanel();
          panel.setLayout(new BoxLayout(panel, BoxLayout.PAGE_AXIS));
          panel.add(new JScrollPane(areaTexto1)); <-
          panel.add(Box.createVerticalStrut(20));
          panel.add(new JScrollPane(areaTexto2));
        add(panel);
    }
     public static void main(String args[]) {
          ModeloCompartido aplicacion = new ModeloCompartido();
        aplicacion.setTitle("Prueba documento compartido");
        aplicacion.setSize(500, 300);
        aplicacion.setVisible(true);
    }
}//ModeloDocumentoCompartido
```

Las áreas de texto se presentan en paneles con desplazamiento.

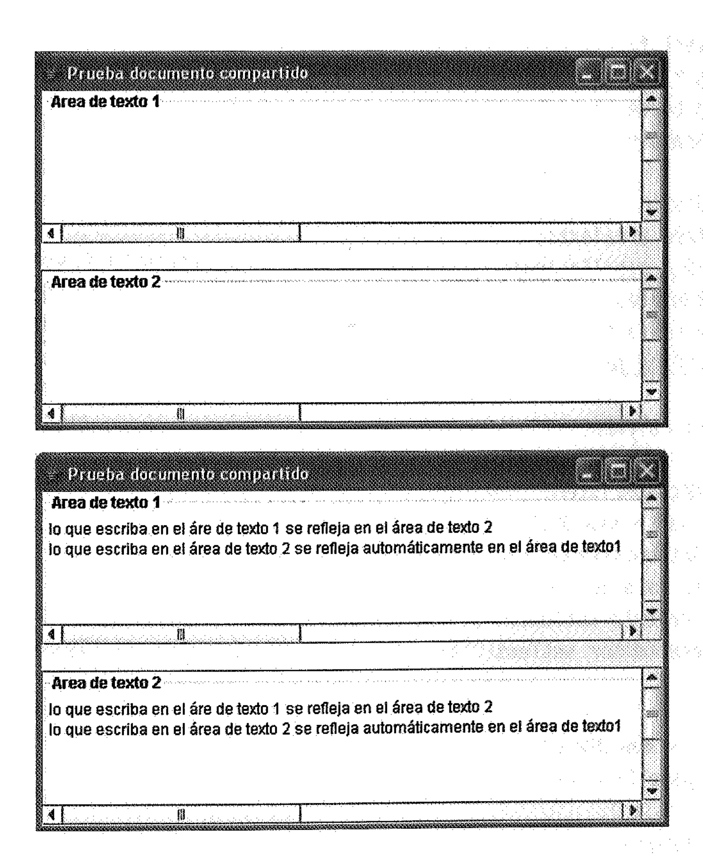


Figura 10.17. Capturas de la aplicación en su estado inicial y después de haber escrito en las dos áreas de texto.

Ejercicio 10.14:

Escriba una aplicación gráfica que permita realizar estadísticas sobre un texto. Debe permitir calcular a elección del usuario el número de caracteres, de palabras y de líneas que contiene. La aplicación contendrá un espacio de edición en el que introducir el texto a analizar y dos botones de operación con los que realizar el cálculo y limpiar el espacio de edición. Los resultados se deben mostrar de forma clara y legible.

Planteamiento: Se crea una ventana con tres zonas, una superior en la que se disponen tres cuadros o casillas de verificación, para que el usuario pueda seleccionar los datos que desea recibir del análisis, y los dos botones de operación con los que realizar el cálculo y limpiar la zona de edición del texto. En la parte central se presentará un área de texto editable para introducir el contenido a analizar. Por si dicho contenido fuera muy extenso se introduce dentro de un panel con barras de desplazamiento (que se fija que sean siempre visibles). En la parte inferior se coloca una zona para mostrar los resultados implementada mediante un JEditorPane que, como es un componente que permite contenido en HTML, se puede hacer una presentación más vistosa.

Los cálculos del número de palabras y del número de líneas se hacen utilizando las facilidades que proporciona la clase java.util.Scanner. En el oyente del botón de cálculo, con ayuda de los métodos privados de cálculo del número de caracteres, palabras y líneas, se construye un texto HTML que usa texto destacado y enfatizado que finalmente se muestra en la zona de resultados.

Para mejorar la legibilidad y la presentación gráfica se han añadido bordes con título a los paneles de edición y de resultados y al panel de elección de estadísticas a realizar. En el botón de limpiar el área de texto editable se ha utilizado HTML para su texto identificativo.

Solución:

```java
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.border.*;
import java.util.Scanner;

public class EstadisticaTexto extends JFrame {
    private JTextArea areaTexto;
    private JCheckBox casillaLineas;
    private JCheckBox casillaPalabras;
    private JCheckBox casillaCaracteres;
    private JEditorPane panelEditor;

    public EstadisticaTexto(){
      JButton calcular = new JButton("Calcular");
      calcular.addActionListener(new OyenteCalcular());
      JButton limpiar = new JButton("<html>Limpiar área <br> de texto");
      limpiar.addActionListener(new ActionListener(){
            public void actionPerformed (ActionEvent evento){
               areaTexto.setText("");
               panelEditor.setText("");
            }
      });
      JPanel panel = new JPanel();
        panel.add(panelEleccion());
      panel.add(calcular);
      panel.add(limpiar);
      add(panel, BorderLayout.NORTH);
        add(panelTexto(), BorderLayout.CENTER);
        add(panelEditor(), BorderLayout.SOUTH);
    }
    private JScrollPane panelTexto(){
      areaTexto= new JTextArea(10,50);
      JScrollPane panelScroll = new JScrollPane(areaTexto);
      panelScroll.setHorizontalScrollBarPolicy(
```

El boton limpiar utiliza texto HTML en su etiqueta.

```
            JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS);
        panelScroll.setVerticalScrollBarPolicy(
                        JScrollPane.VERTICAL_SCROLLBAR_ALWAYS);
        TitledBorder titulo;
        titulo = BorderFactory.createTitledBorder("Texto a analizar");
         panelScroll.setBorder(titulo);
        return panelScroll;
    }

    private JPanel panelEleccion(){
        JPanel panel = new JPanel();
        casillaCaracteres = new JCheckBox("Caracteres", true);
        casillaPalabras = new JCheckBox("Palabras");
        casillaLineas = new JCheckBox("Lineas");
         panel.add(casillaCaracteres);
         panel.add(casillaPalabras);
        panel.add(casillaLineas);
        TitledBorder titulo;
         titulo = BorderFactory.createTitledBorder("Datos deseados");
        panel.setBorder(titulo);
        return panel;
    }

    private JEditorPane panelEditor(){
        panelEditor = new JEditorPane();
        panelEditor.setEditable(false);
        panelEditor.setContentType("text/html");
         panelEditor.setPreferredSize(new Dimension(200, 125));
         panelEditor.setMinimumSize(new Dimension(10, 10));
        TitledBorder titulo;
        titulo = BorderFactory.createTitledBorder("Resultados (texto html)");
         panelEditor.setBorder(titulo);
        return panelEditor;
    }

    private int calcularCaracteres(String cadena){
        return cadena.length();
    }

    private int calcularPalabras(String cadena){
        Scanner scaner = new Scanner(cadena);
        int palabras = 0;
        while (scaner.hasNext()){
            scaner.next();
            palabras++;
        }
        return palabras;
    }
    private int calcularLineas(String cadena){
        Scanner scaner = new Scanner(cadena);
        int lineas = 0;
```

Las barras de desplazamiento del panel que contiene al área de texto se mostrarán siempre.

Se fija que el contenido del panel editor sea HTML y se establecen sus tamaños preferido y mínimo.

```
        while (scaner.hasNextLine()){
            scaner.nextLine();
            lineas++;
        }
        return lineas;
    }

    class OyenteCalcular implements ActionListener{
        public void actionPerformed (ActionEvent evento){
            String texto=areaTexto.getText();
            StringBuffer resultado = new StringBuffer("<html>");
            if (casillaCaracteres.isSelected()){
                resultado.append("<strong>Caracteres: </strong>");
                resultado.append("<em>"+ calcularCaracteres(texto) + "</em> <br>") ;
            }
            if (casillaPalabras.isSelected()){
                resultado.append("<strong>Palabras: </strong>");
                resultado.append("<em>"+ calcularPalabras(texto) + "</em> <br>");
            }
            if (casillaLineas.isSelected()){
                resultado.append("<strong>Líneas: </strong>");
                resultado.append("<em>"+ calcularLineas(texto)+ "</em> <br>");
            }
            panelEditor.setText(resultado.toString()+ "</html>");
        }
    }

    public static void main(String args[]) {
        EstadisticaTexto aplicacion = new EstadisticaTexto();
        aplicacion.setTitle("Estadistica Texto");
        aplicacion.pack();
         aplicacion.setVisible(true);
    }
}
```

En resultado se va construyendo un texto HTML que es el que se muestra en el panel de resultados.

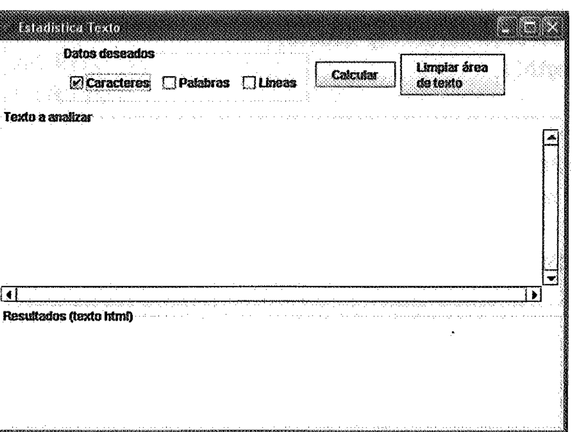


Figura 10.18. Captura inicial de la aplicación. Fíjese cómo la casilla de verificación de caracteres aparece seleccionada y cómo el botón de limpiar área de texto usa texto HTML que ocupa dos líneas.

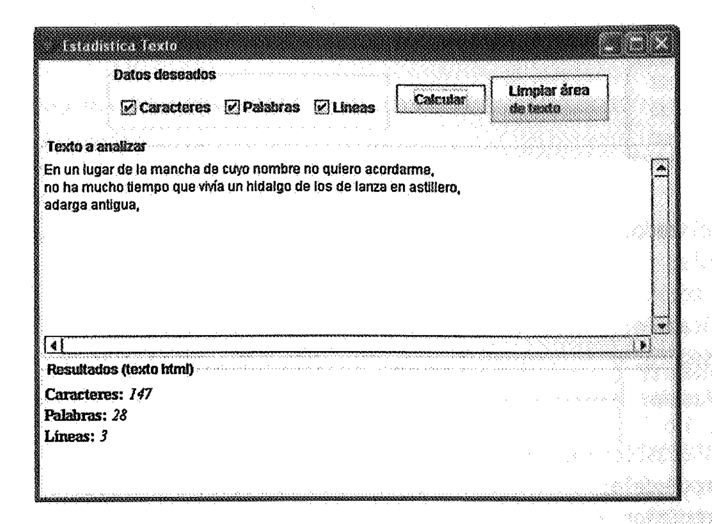


Figura 10.19. Capturas de la aplicación después de haber escrito un texto y haber pulsado el botón Calcular. En el campo de resultados, al utilizar HTML, parte del texto aparece en negrita y los resultados numéricos en cursiva.

Comentario: En el componente `JEditorPane` se puede visualizar contenido en HTML, texto plano o documentos en formato RTF (*Rich Text Format*). Estos campos son muy versátiles ya que el texto mostrado se puede hacer todo lo que el formato admite como, por ejemplo, permitir de forma simple visualizar texto con colores, con distintos tipos de fuente, incorporando imágenes, etc.

Ejercicio 10.15:

Escriba una aplicación gráfica que presente un panel en el que se contemplen los datos de la clase Coche que se presentó en el Ejercicio 2.1 del Capítulo 2 sobre Clases y objetos. En este panel se deben poder editar los datos del objeto coche.

Planteamiento: El panel debe poder representar todos los atributos de la clase `Coche` de una forma clara y visual. Con este propósito a todos los atributos de tipo `String` se les proporciona un campo de texto en el que el usuario pueda introducir datos de forma libre, excepto con el año de fabricación ya que este dato sólo es válido si es numérico y por tanto se usa un campo de texto con formato (`JFormattedText`) que limita su contenido a números. Para los campos enumerados se ha optado por dos soluciones diferentes. En el caso del seguro, como sólo hay dos posibles valores, se ha decidido utilizar botones de elección mutuamente excluyentes. En el caso del tipo de coche, como el número de valores es mayor, se ha elegido un cuadro combinado desplegable (`JComboBox`). En el caso del tipo de pintura, se ha incluido una casilla de verificación que se marca en el caso de que la pintura sea metalizada.

El gestor de disposición elegido es `GridLayout` ya que permite la organización sencilla en una matriz de dos columnas de los elementos de interacción y de sus correspondientes etiquetas explicativas. Para mejorar la agrupación visual de todos los elementos se le añade un borde con título.

Solución:

```
import javax.swing.*;
import java.awt.*;
import javax.swing.text.*;
import java.text.*;
import javax.swing.border.*;

enum TipoDeCoche { MINI, UTILITARIO, FAMILIAR, DEPORTIVO };
```

```
enum TipoDeSeguro { A_TERCEROS, A_TODO_RIESGO };

class Coche {

    String modelo;
    String color;
    boolean esMetalizado;
    String matricula;
    TipoDeCoche tipo;
    int añoDeFabricación;
    TipoDeSeguro seguro;
}

public class PanelDatosCoche extends JPanel {
    JTextField campoModelo;
    JTextField campoColor;
    JCheckBox esMetalizada;
    JTextField campoMatricula;
    JComboBox tipoCoche;
    JFormattedTextField campoAño;
    JRadioButton todoRiesgo;
    JRadioButton terceros;

    public PanelDatosCoche() {
      setLayout(new GridLayout(7,2));
      JLabel modelo = new JLabel("Modelo: ", JLabel.RIGHT);
      campoModelo = new JTextField();
      add(modelo);
      add(campoModelo);

        JLabel color = new JLabel("Color: ", JLabel.RIGHT);
      campoColor = new JTextField();
      add(color);
      add(campoColor);

        JLabel matricula = new JLabel("Matricula: ", JLabel.RIGHT);
      campoMatricula = new JTextField();
      add(matricula);
      add(campoMatricula);

      ButtonGroup grupoBotones= new ButtonGroup();
      todoRiesgo = new JRadioButton("A todo riesgo", true);
      todoRiesgo.setMnemonic(KeyEvent.VK_R);
      terceros = new JRadioButton("A terceros");
      terceros.setMnemonic(KeyEvent.VK_T);
      grupoBotones.add(todoRiesgo);
      grupoBotones.add(terceros);
      add(todoRiesgo);
      add(terceros);

      JLabel año = new JLabel("Año de fabricación : ", JLabel.RIGHT);
```

Para mejorar la asociación visual de la etiqueta al campo se pone su alineación horizontal a la derecha.

```
        MaskFormatter formato = null;
        try{
            formato = new MaskFormatter("####");
        } catch (ParseException e){
            //se captura la excepción y no se hace nada
        }
        campoAño = new JFormattedTextField(formato);
        add(año);
        add(campoAño);

        JLabel tipo = new JLabel("Tipo de coche : ", JLabel.RIGHT);
        tipoCoche = new JComboBox(TipoDeCoche.values());
        add(tipo);
        add(tipoCoche);

        JLabel pintura = new JLabel("Tipo de pintura : ", JLabel.RIGHT);
        esMetalizada = new JCheckBox("Metalizada", false);
        add(pintura);
        add(esMetalizada);

        TitledBorder titulo;
        titulo = BorderFactory.createTitledBorder("Datos del coche");
        setBorder(titulo);
    }

    public static void main(String[] args) {
        JFrame ventana = new JFrame("Panel de datos");
        ventana.add(new PanelDatosCoche());
        ventana.pack();
        ventana.setVisible(true);
    }
}
```

Mascara para que el formato del campo año sólo admita hasta cuatro dígitos numéricos.

El cuadro combinado tendrá como valores los del tipo enumerado TipoDeCoche.

Inicialmente la casilla aparecerá como no marcada.

Figura 10.20. Capturas del panel de datos en su estado original, con el cuadro de selección JComboBox desplegado y una vez con los datos rellenos.

Comentario: Fíjese que aunque en este caso puede parecer muy adecuado poner el tipo de seguro como botones de selección excluyentes, si el número de valores aumenta se debería volver a diseñar la interfaz gráfica. Sin embargo cuando se usa un cuadro de selección combinado como en tipo de coche, dicha modificación aparecería automáticamente en la interfaz sin necesidad de nueva programación.

Ejercicio 10.16:

Utilizando el concepto de documento compartido que se presentó en el Ejercicio 10.13 escriba una aplicación que permita simular en una única computadora mediante varias ventanas un sistema sencillo de conversación en la red, es decir, un chat. Para simplificar su utilización por parte del usuario esta aplicación debe incluir menús de operación y debe proporcionar un acceso múltiple a la operación de envio de mensajes al foro de conversación. También permitirá configurar que se identifique o no el usuario que envia el mensaje.

Planteamiento: La idea principal es hacer una aplicación que permita configurar el modelo de documento que se utiliza para el área de conversación, de modo que se pueda compartir entre las diferentes ventanas. En este caso la clase principal `SimulacionChat` no extiende ninguna clase gráfica, sólo las utiliza. En la ventana principal de la aplicación se incluyen un área de texto para la conversación, un campo de texto para el mensaje que se envía y un botón `Enviar` que hace que el contenido del campo de texto se añada a la conversación. Además se incluye una barra de menús con dos menús, uno de operaciones que permite borrar la conversación, enviar el mensaje o salir de la aplicación y otro menú de configuración que permite determinar la identificación y obtener información sobre el programa que aparecerá en un cuadro emergente. La gestión de la identificación se hace mediante un submenú que contiene dos botones de elección para configurar si el mensaje se envía de forma anónima o bien identificado.

En este caso se permite el envío de mensajes de tres formas alternativas. Pulsando Intro en el campo de texto del mensaje, haciendo clic en el botón `Enviar` o bien seleccionando la opción `Enviar mensaje` del menú `Operaciones`.

Solución:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.text.*;
import javax.swing.border.*;

public class SimulacionChat {
    final int ANCHO = 40;
    String identidad; <-
    boolean identificado;
    JTextArea areaTexto;
    JTextField campoTexto;
    JFrame ventana;

    public SimulacionChat( Document documento, String identidad){
      identificado= false;
      this.identidad = identidad;
      ventana = new JFrame("Simulacion Chat: " + identidad);
      ventana.setJMenuBar(crearMenu());
      areaTexto= new JTextArea(documento);
      areaTexto.setColumns(ANCHO);
      areaTexto.setRows(15);
      areaTexto.setBorder(BorderFactory.createTitledBorder("Conversación"));
      OyenteTexto oyente = new OyenteTexto(this);
```

> Los atributos identidad e identificado definen la identificación de cada usuario y si se debe mostrar junto con el mensaje enviado al chat.

```
        JPanel panel = new JPanel();
        campoTexto = new JTextField(ANCHO);
        campoTexto.setBorder(BorderFactory.createTitledBorder("Mensaje"));
        campoTexto.addActionListener(oyente);
        JButton enviar = new JButton("Enviar");
        enviar.addActionListener(oyente);
        panel.add(campoTexto);
        panel.add(enviar);
        Container panelContenido = ventana.getContentPane();
        panelContenido.setLayout(new BoxLayout(panelContenido, BoxLayout.PAGE_AXIS));
        panelContenido.add(new JScrollPane(areaTexto));
        panelContenido.add(Box.createGlue());
        panelContenido.add(panel);
        panelContenido.add(Box.createGlue());
        ventana.pack();
        ventana.setVisible(true);
    }

    public SimulacionChat(String identidad){
        this(null, identidad);
    }
    private JMenuBar crearMenu(){
        JMenuBar barraMenu= new JMenuBar();
         JMenu operaciones = new JMenu("Operaciones");
         operaciones.setMnemonic(KeyEvent.VK_O);
         JMenuItem borrar = new JMenuItem("Borrar coversación", KeyEvent.VK_B);
         JMenuItem enviar = new JMenuItem("Enviar mensaje", KeyEvent.VK_E);
         JMenuItem salir= new JMenuItem("Salir", KeyEvent.VK_S);
         operaciones.add(borrar);
         operaciones.add(enviar);
         operaciones.add(new JSeparator());
         operaciones.add(salir);
        borrar.addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent e) {
                areaTexto.setText("");
            }
        });
        enviar.addActionListener(new OyenteTexto(this));
        salir.addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent e) {
                System.exit(0);
            }
        });
        JMenu configuracion = new JMenu("Configuración");
        configuracion.setMnemonic(KeyEvent.VK_C);
        JMenu identidad = new JMenu("Identidad");
        ButtonGroup grupoIdentidad = new ButtonGroup();
        JRadioButtonMenuItem anonimo = new JRadioButtonMenuItem("Anónimo", true);
        JRadioButtonMenuItem identificado = new JRadioButtonMenuItem("Identificado");
        anonimo.addActionListener(new OyenteIdentidad());
        identificado.addActionListener(new OyenteIdentidad());
```

Invocación al otro constructor.

Crea la barra de menú de la aplicación.

Crea la línea separadora entre las operaciones normales y salir de la aplicación.

Submenú que gestiona el control de si se envían los mensajes identificados o no.

```
        grupoIdentidad.add(anonimo);
        grupoIdentidad.add(identificado);
        identidad.add(anonimo);
        identidad.add(identificado);
        JMenuItem acerca = new JMenuItem("Acerca", KeyEvent.VK_A);
        acerca.addActionListener(new OyenteAcerca());
        configuracion.add(identidad);
        configuracion.add(new JSeparator());
        configuracion.add(acerca);
        barraMenu.add(operaciones);
        barraMenu.add(configuracion);
         return barraMenu;
    }// crearMenu

    public Document devuelveDocumento(){
       return areaTexto.getDocument();
    }

     class OyenteAcerca implements ActionListener{
       public void actionPerformed(ActionEvent e) {
          JOptionPane.showMessageDialog(null,
             "Consulte también el libro\n" +
             "Java 2: iniciación y referencia. McGraw-Hill",
             "Acerca de este programa",
             JOptionPane.INFORMATION_MESSAGE);
       }
    }//OyenteAcerca

    class OyenteIdentidad implements ActionListener{
       public void actionPerformed(ActionEvent e) {
          JRadioButtonMenuItem boton = (JRadioButtonMenuItem)e.getSource();
          if ("Anonimo".equals(boton.getText())){
             identificado = false;
          }else{
             identificado = true;
          }
       }
    }

    public static void main(String args[]) {
       SimulacionChat aplicacion = new SimulacionChat("Prueba");
    }
}//SimulacionChat

class OyenteTexto implements ActionListener {
    SimulacionChat aplicacion;
    public OyenteTexto(SimulacionChat aplicacion){
        this.aplicacion= aplicacion;
    }
    public void actionPerformed(ActionEvent evento){
      JTextArea  areaTexto=aplicacion.areaTexto;
```

Metodo que devuelve el modelo de documento del área de texto donde se almacena la conversación.

Se comprueba si se ha marcado el botón anónimo o el identificado del menú para fijar el estado de identificación.

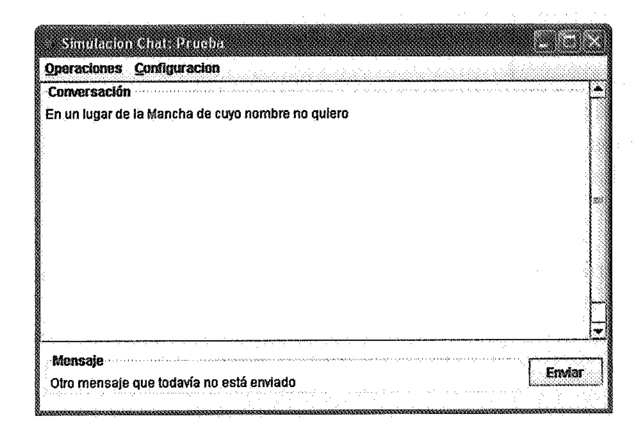
Este programa principal permite probar por separado el funcionamiento básico de la clase `SimulacionChat`.

Oyente de las operaciones de envio de texto desde el campo de texto, desde el boton enviar y desde la opción de menú. Es una clase externa.

```
            if (aplicacion.identificado){
               areaTexto.append(aplicacion.identidad+" > ");
            }
            areaTexto.append(aplicacion.campoTexto.getText()+"\n");
            aplicacion.campoTexto.setText("");
        }
    } // OyenteTexto

    class Servidor {
        public static void main(String args[]) {
            SimulacionChat cliente1 = new SimulacionChat("Cliente1");
            Document documento = cliente1.devuelveDocumento();
            SimulacionChat cliente2 = new SimulacionChat(documento,"Cliente2");
            SimulacionChat cliente3 = new SimulacionChat(documento,"Cliente3");
        }
    } // Servidor
```

Clase que permite comprobar el funcionamiento completo de la simulación lanzando 3 ventanas que comparten el modelo de documento.



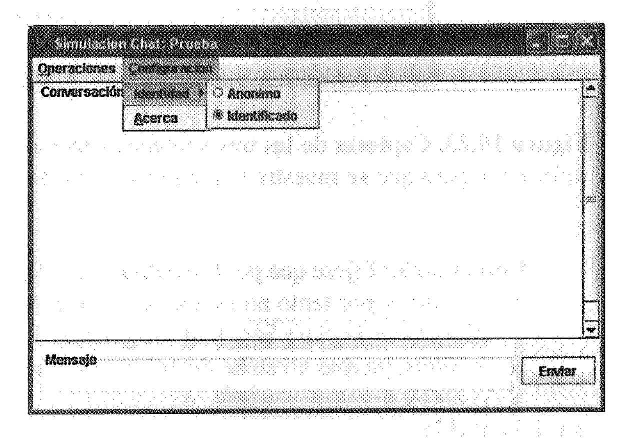
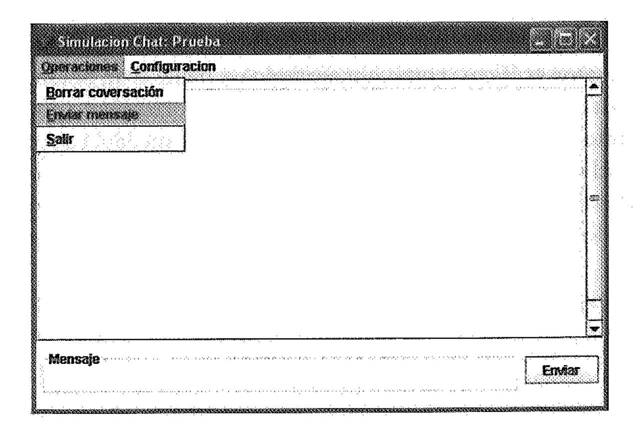


Figura 10.21. Captura inicial de la aplicación en la que se ha enviado ya un mensaje de forma anónima y se va a enviar otro. A la derecha mediante el menú se cambia la configuración para que los mensajes aparezcan identificados. En este caso se ha ejecutado el `main()` incluido en la clase `SimulacionChat`.



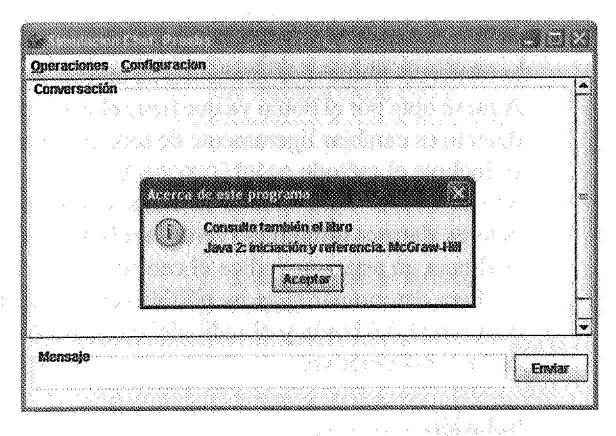


Figura 10.22. El mensaje se puede enviar también mediante la opción de menú `Enviar mensaje`. A la derecha se muestra el cuadró de diálogo con información sobre el programa que se muestra seleccionando la opción `Acerca` del menú `Configuración`.

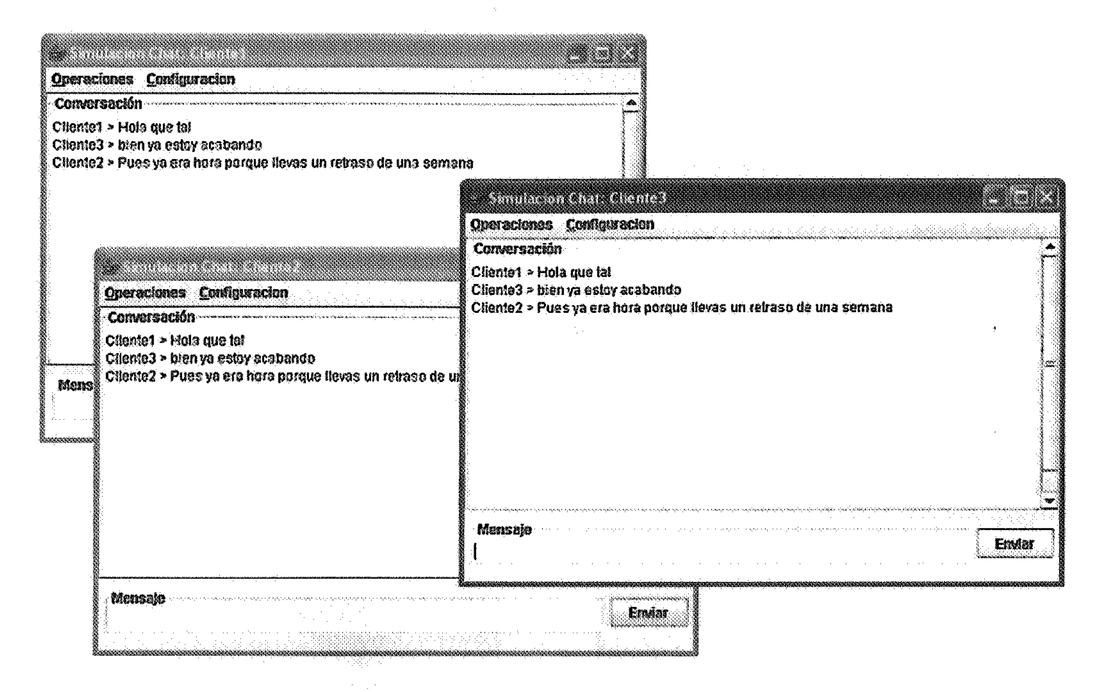


Figura 10.23. Capturas de las tres ventanas generadas y de los mensajes intercambiados habiendo configurado la aplicación para que se muestre la identidad. En este caso se ha ejecutado el `main()` incluido en la clase `Servidor`.

Comentario: Fíjese que por brevedad en el código los atributos de la clase `SimulacionChat` no se han declarado privados y por tanto no es necesario proporcionar los métodos de acceso correspondientes.

Al ejecutar la clase `Servidor` se generan tres ventanas de la aplicación y salen superpuestas, unas encima de las otras, ya que no se ha determinado ninguna posición inicial de las ventanas.

Ejercicio 10.17:

Escriba una aplicación gráfica que presente una diana de modo que el usuario pueda interactuar con ella haciendo clic con el ratón. Se informará al usuario si se ha acertado en el centro de dicha diana o no. La diana debe poderse cambiar de tamaño y no tiene que ser necesariamente circular, pudiendo ser ovalada.

Planteamiento: Para representar una diana y que se pueda interactuar con ella inicialmente se podría cambiar la forma de dibujo o presentación de alguno de los elementos gráficos, como por ejemplo un panel o un botón. Aquí se opta por el botón ya que tiene el concepto de pulsación directamente asociado y su comportamiento por defecto es cambiar ligeramente de tono de color. Por tanto se hace que la clase `Diana` extienda a un `JButton` y se incluye el método `paintComponent(Graphics g)` en el cual se decide cuál será la apariencia de este nuevo tipo de botón. Para simular la diana se dibujan círculos u óvalos cada vez más pequeños que se rellenan de colores alternos diferentes (verde, azul). El círculo u óvalo central se rellena de color rojo. Mediante dos líneas se dibuja un aspa que indica el centro real del botón.

Para determinar si se ha pulsado en el círculo u óvalo central de la diana se calcula la distancia entre el centro real del botón y el radio del círculo u óvalo central y en función del resultado se muestra un cuadro de diálogo informativo.

Solución:

```
import javax.swing.*;
import java.awt.*;
import javax.swing.event.*;
```

```
import java.awt.event.*;

public class Diana extends JButton {
    static final int NUM=5;
    int centroX;
    int centroY;

    public Diana(){
        this.setBackground(Color.cyan);
        this.setPreferredSize(new Dimension(250,250));

        this.addMouseListener(new MouseInputAdapter(){
         public void mousePressed(MouseEvent e) {
            int x = e.getX();
            int y = e.getY();
            String mensaje = "No has dado en el centro";
            if (((Math.abs(x-centroX))<(centroX/NUM))&&
               ((Math.abs(y-centroY))<(centroY/NUM))){
               mensaje = "Muy bien acertaste en el centro";
            }
            JOptionPane.showMessageDialog(Diana.this,
               mensaje,
               "Pulsación del ratón realizada",
               JOptionPane.INFORMATION_MESSAGE);
         }
       });
   }//constructor

   public void paintComponent(Graphics g) {
      super.paintComponent(g); //pintado del fondo
      int alto = getSize ().height;
      int ancho = getSize ().width;
      centroX = ancho/2;
      centroY = alto/2;
      for(int i = NUM; i > 0; i--){
         if (i%2 == 0) {
            g.setColor(Color.green);
         }else{
            g.setColor(Color.blue);
         }
         if (i == 1){
            g.setColor(Color.red);
         }
         int radioX = i * centroX/NUM;
         int radioY = i * centroY/NUM;
         g.fillOval(centroX-radioX, centroY-radioY, 2*radioX, 2*radioY);
      }
      g.setColor(Color.black);
      g.drawLine(0, 0, ancho, alto);
      g.drawLine(0, alto, ancho, 0);
   }//paintComponent
```

Se fija el color de fondo (azul claro) y el tamaño preferido de la diana.

Metodo que permite modificar la forma en la que se presenta o se dibuja un componente gráfico.

Se cambia el color que se utilizará para rellenar el círculo u ovalo.

Se dibujan y rellenan los círculos u óvalos.

Se dibuja el aspa.

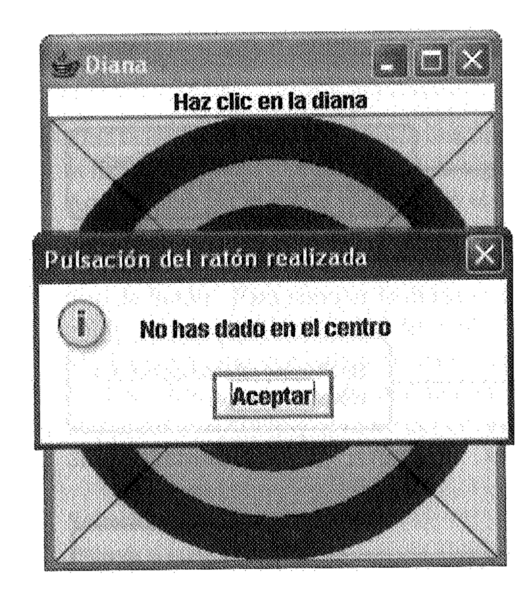
```
    public static void main(String args[]) {
        JFrame ventana = new JFrame("Diana");
        Diana diana = new Diana();
        ventana.add(diana);
        JLabel etiqueta = new JLabel("Haz clic en la diana", JLabel.CENTER);
        ventana.add(etiqueta, BorderLayout.NORTH);
        ventana.pack();
        ventana.setVisible(true);
    }
}
```

Comentario: Fíjese que como el botón puede no ser cuadrado no se puede calcular simplemente la distancia al centro para saber si se ha pulsado en el círculo central, si no que hay que hacerlo por separado para la coordenada X y para la coordenada Y.





Figura 10.24. Capturas de la diana con su forma original cuadrada y una vez que se ha redimensionado la ventana.



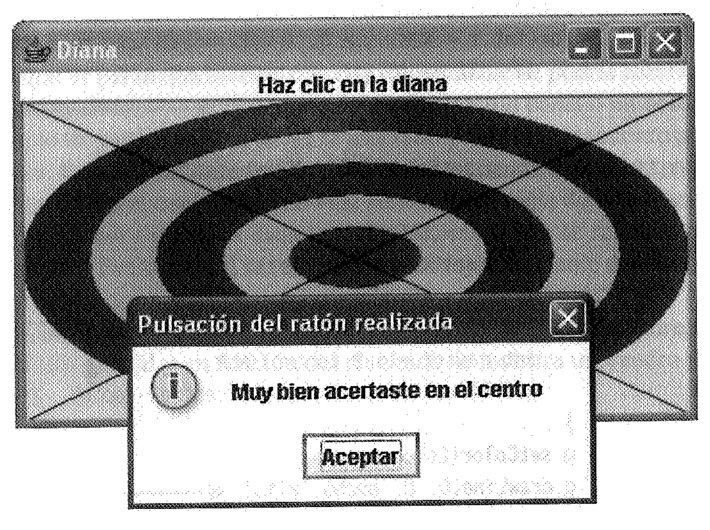


Figura 10.25. Capturas de la interacción del usuario indicándole si ha acertado en el centro o no.

Ejercicio 10.18:

Escriba una aplicación gráfica que reutilizando el panel de datos realizado en el Ejercicio 10.15 permita gestionar una lista de coches. Inicialmente las operaciones que permitirá dicha lista serán añadir nuevos coches y listar los coches existentes.

Planteamiento: La aplicación se plantea como un conjunto de clases con una clase `InterfazGrafica` que crea la ventana principal que contiene una barra de menús, una lista donde se presentan los coches introducidos (`JList`) y dos botones que permiten añadir un nuevo coche y visualizar los coches existentes. Cuando se hace clic en el botón `Añadir` se presenta un cuadro emergente de entrada de datos que presenta el panel del coche en el cual el usuario puede rellenar todos los datos. En este caso se incluye un completo control de errores de modo que no se permite que se pueda crear un objeto coche que no sea correcto. Para la entrada de datos, para informar de los errores o para informar al usuario que la operación se ha cancelado, se hace mediante cuadros de diálogo aprovechando las funcionalidades ofrecidas por la clase `JOptionPane`. Todos los oyentes, para proporcionar el comportamiento de la aplicación, se han desarrallado como clases internas o como clases anónimas.

Se reutilizan las clases existentes realizando cambios mínimos donde ha sido necesario. En la enumeración `TipoDeCoche`, como es una clase, se añade un método `getTipo()` para simplificar la lectura de datos que a partir de una cadena de texto con el valor del tipo enumerado (tanto en mayúsculas como en minúsculas) devuelve el correspondiente valor. En la clase `Coche` se incluye el constructor y el método `toString()` que utilizando las características de formato permite presentar directamente un coche en una lista (`JList`).

Se ha creado una nueva clase `Aplicacion` que es la encargada de gestionar la lista de coches. Esta es una clase muy básica que tiene como atributo una lista genérica de `Coches` que posteriormente se instancia en el constructor con un `ArrayList`. En este ejemplo se puede observar cómo se le añade una interfaz gráfica a una aplicación preexistente sin necesidad de mezclar el código de interacción y presentación con el código de operación de la aplicación. Fíjese que la aplicación es completamente independiente de su interfaz gráfica y, de hecho, no necesita disponer de ninguna referencia a dicha interfaz (la interfaz gráfica sí dispone de una referencia a la aplicación como se describe a continuación).

Finalmente se añade una clase `EjecutarAplicacion` que relaciona la aplicación con la interfaz gráfica para lo cual incluye un método principal en el que se crean dos objetos concretos de las clases `Aplicación` e `InterfazGrafica`. Por tanto para ejecutar la aplicación completa se debe utilizar `EjecutarAplicacion` como clase principal.

Por brevedad se presentan agrupados al principio del código todos los paquetes necesarios para el desarrollo de todas las clases.

Solución:

```
import javax.swing.*;
import java.awt.*;
import javax.swing.text.*;
import java.text.*;
import javax.swing.border.*;
import java.util.*;
import java.awt.event.*;

enum TipoDeCoche {
    MINI, UTILITARIO, FAMILIAR, DEPORTIVO;

    // devuelve el enumerado a partir de su nombre
    public static TipoDeCoche getTipo(String cadena){  ← Método que simplifica la lectura del tipo enumerado.
        return Enum.valueOf(TipoDeCoche.class, cadena.toUpperCase());
    }
}//TipoDeCoche
```

```
enum TipoDeSeguro { A_TERCEROS, A_TODO_RIESGO };

class Coche {
    String modelo;
    String color;
    boolean esMetalizado;
    String matricula;
    TipoDeCoche tipo;
    int añoDeFabricacion;
    TipoDeSeguro seguro;

    public Coche(String modelo, String color, boolean esMetalizado, String matricula,
              TipoDeCoche tipo, int año, TipoDeSeguro seguro){
      this.modelo= modelo;
      this.color= color;
      this.esMetalizado= esMetalizado;
      this.matricula = matricula;
      this.tipo = tipo;
      this.añoDeFabricacion = año;
      this.seguro = seguro;
    }

    public String toString(){
      Formatter formato = new Formatter();
      formato.format("- %10s - %8s - %5d - %6s - %7s - metalizada: %5b - %8s",
      modelo, color, añoDeFabricacion, tipo, matricula, esMetalizado, seguro);
      return formato.toString();
    }
}//Coche

class Aplicacion{
    private java.util.List<Coche> lista;

    public Aplicacion(){
        lista = new ArrayList<Coche>();
    }

    public void añadirCoche(Coche coche){
        lista.add(coche);
    }

    public java.util.List<Coche> obtenerLista(){
        return lista;
    }
}// Aplicacion

class InterfazGrafica extends JFrame{
    private Aplicacion aplicacion;
    private JButton añadir;
    private JButton visualizar;
    private DefaultListModel modeloLista;
```

Se realiza una salida con formato en la conversión del objeto a cadena.

La lista general se instancia con un `ArrayList` como lista concreta.

```
OyenteVisualizar oyenteVisualizar;

public InterfazGrafica(Aplicacion aplicacion){
   this.aplicacion = aplicacion;
   oyenteVisualizar = new OyenteVisualizar();
   JPanel panel = new JPanel();
   panel.setLayout(new BoxLayout(panel, BoxLayout.PAGE_AXIS));
   Border borde = BorderFactory.createEmptyBorder(10,10,10,10);
   TitledBorder titulo = BorderFactory.createTitledBorder(borde, "Lista coches",
            TitledBorder.CENTER, TitledBorder.TOP);
    panel.setBorder(titulo);
    modeloLista = new DefaultListModel();
    JList lista = new JList(modeloLista);
    panel.add(new JScrollPane(lista));

   this.setLayout(new BoxLayout(this.getContentPane(), BoxLayout.PAGE_AXIS));
   add(panel);
   add(panelBotones());

   setJMenuBar(creaMenus());
   setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
   setTitle("Gestion de coches");
   setSize(300,250);
   setVisible(true);

}//constructor

private JPanel panelBotones(){
   JPanel panel = new JPanel();
   panel.setLayout(new BoxLayout(panel, BoxLayout.LINE_AXIS));
   panel.setBorder(new EmptyBorder(0,10,10,10));
   añadir = new JButton("Añadir");
   añadir.setMnemonic(KeyEvent.VK_A);
   añadir.addActionListener(new OyenteNuevo());
   visualizar = new JButton("Visualizar");
   visualizar.setMnemonic(KeyEvent.VK_V);
   visualizar.addActionListener(new OyenteVisualizar());
   panel.add(Box.createHorizontalGlue());
   panel.add(añadir);
   panel.add(Box.createRigidArea(new Dimension(10,0)));
   panel.add(visualizar);
   return panel;
}//panelBotones

private JMenuBar creaMenus(){
   JMenuBar barraMenu = new JMenuBar();
   JMenu menuOperaciones = new JMenu("Operaciones");
   menuOperaciones.setMnemonic(KeyEvent.VK_O);
   JMenuItem listar = new JMenuItem("Listar coches", KeyEvent.VK_L);
   listar.addActionListener(oyenteVisualizar);
   menuOperaciones.add(listar);
```

Se crea un borde vacío y un borde con título centrado en la parte superior.

```
    menuOperaciones.add(new JSeparator());
    JMenuItem salir= new JMenuItem("Salir", KeyEvent.VK_S);
    salir.addActionListener(new ActionListener(){
          public void actionPerformed(ActionEvent e) {
             System.exit(0);
          }
       });
    menuOperaciones.add(salir);
    barraMenu.add(menuOperaciones);
    return barraMenu;
}//creaMenus

class OyenteVisualizar implements ActionListener{
   public void actionPerformed(ActionEvent evento){
      modeloLista.clear();
      Iterator iterador = aplicacion.obtenerLista().listIterator();
      while(iterador.hasNext())
            modeloLista.addElement(iterador.next());
   }
}//OyenteVisualizar

class OyenteNuevo implements ActionListener{
   public void actionPerformed(ActionEvent evento){
      boolean error = false;
      PanelDatosCoche panel = new PanelDatosCoche();
      if (JOptionPane.showConfirmDialog(InterfazGrafica.this
                     ,panel
                     ,"Introduzca datos"
                     ,JOptionPane.OK_CANCEL_OPTION
                     ,JOptionPane.PLAIN_MESSAGE
                     ) == JOptionPane.OK_OPTION) {
         String modelo = panel.campoModelo.getText();
         String color = panel.campoColor.getText();
         Boolean esMetalizado = panel.esMetalizada.isSelected();
         String matricula = panel.campoMatricula.getText();
         TipoDeCoche tipo = TipoDeCoche.getTipo(panel.tipoCoche.getSelectedItem().toString());
         int año= 0;
         try {
            año =Integer.parseInt(panel.campoAño.getText());
         }catch (NumberFormatException e){
            error = true;
            JOptionPane.showMessageDialog(null,
               "<html><u>El año con cuatro dígitos (e.g. 2003)",
               "Error",
               JOptionPane.ERROR_MESSAGE);
         }
         error = error || (modelo.length() ==0) ||
                (color.length() ==0)|| (matricula.length() ==0);
         if (error)
            JOptionPane.showMessageDialog(null,
               "<html>Tipo de error "+
```

Sincroniza el contenido de la JList con los coches de la aplicación utilizando el modelo de la lista.

El panel de los datos del coche se presenta dentro del cuadro de diálogo.

```
                    "<p color='red'>Algún campo está vacío"+
                    " <em> Debe rellenarlos todos",
                    "Error",
                    JOptionPane.ERROR_MESSAGE);
                TipoDeSeguro seguro;
                if (panel.todoRiesgo.isSelected())
                    seguro = TipoDeSeguro.A_TODO_RIESGO;
                else
                    seguro = TipoDeSeguro.A_TERCEROS;
                if (!error) {
                    Coche coche = new Coche(modelo, color, esMetalizado,
                    matricula, tipo, año, seguro);
                    aplicacion.añadirCoche(coche);
                }
            }else{
                JOptionPane.showMessageDialog(null,
                                "Operación no realizada",
                                "Aviso",
                                JOptionPane.WARNING_MESSAGE);
            }

        }
    }//OyenteNuevo

}//InterfazGrafica

class PanelDatosCoche extends JPanel {
    JTextField campoModelo;
    JTextField campoColor;
    JCheckBox esMetalizada;
    JTextField campoMatricula;
    JComboBox tipoCoche;
    JFormattedTextField campoAño;
    JRadioButton todoRiesgo;
    JRadioButton terceros;

    public PanelDatosCoche() {
      setLayout(new GridLayout(7,2));
      JLabel modelo = new JLabel("Modelo: ", JLabel.RIGHT);
      campoModelo = new JTextField();
      add(modelo);
      add(campoModelo);

        JLabel color = new JLabel("Color: ", JLabel.RIGHT);
      campoColor = new JTextField();
      add(color);
      add(campoColor);

        JLabel matricula = new JLabel("Matricula: ", JLabel.RIGHT);
      campoMatricula = new JTextField();
      add(matricula);
```

En los mensajes de los diálogos de error se usa texto HTML.

```
        add(campoMatricula);

        ButtonGroup grupoBotones= new ButtonGroup();
        todoRiesgo = new JRadioButton("A todo riesgo", true);
        todoRiesgo.setMnemonic(KeyEvent.VK_R);
        terceros = new JRadioButton("A terceros");
        terceros.setMnemonic(KeyEvent.VK_T);
        grupoBotones.add(todoRiesgo);
        grupoBotones.add(terceros);
        add(todoRiesgo);
        add(terceros);

        JLabel año = new JLabel("Año de fabricación : ", JLabel.RIGHT);
        MaskFormatter formato=null;
        try{
           formato = new MaskFormatter("####");
        } catch (ParseException e){
           //se captura la excepción y no se hace nada
        }
        campoAño = new JFormattedTextField(formato);
        add(año);
        add(campoAño);

        JLabel tipo = new JLabel("Tipo de coche : ", JLabel.RIGHT);
        tipoCoche = new JComboBox(TipoDeCoche.values());
        add(tipo);
        add(tipoCoche);

        JLabel pintura = new JLabel("Tipo de pintura : ", JLabel.RIGHT);
        esMetalizada = new JCheckBox("Metalizada", false);
        add(pintura);
        add(esMetalizada);

        TitledBorder titulo;
        titulo = BorderFactory.createTitledBorder("Datos del coche");
        setBorder(titulo);
    }

    public static void main(String[] args) {
        JFrame ventana = new JFrame("Panel de datos");
        ventana.add(new PanelDatosCoche());
        ventana.pack();
        ventana.setVisible(true);
    }
}//PanelDatosCoche

public class EjecutarAplicacion {
    public static void main(String args[]){
        Aplicacion aplicacion = new Aplicacion();
        InterfazGrafica ventana = new InterfazGrafica(aplicacion);
    }
} //EjecutarAplicacion
```

Comentario: Fíjese que la operación de `Visualizar` que sincroniza el contenido de la lista de coches y su visualización en el panel con desplazamiento no sería estrictamente necesaria. Dicha actualización se hace mediante el modelo de contenido de la lista `JList` y podría hacerse directamente cada vez que se añadiera un nuevo coche en la lista. En una aplicación más sofisticada, el panel con los datos del coche se podría haber reutilizado además para la presentación y modificación de los datos de un coche en particular.

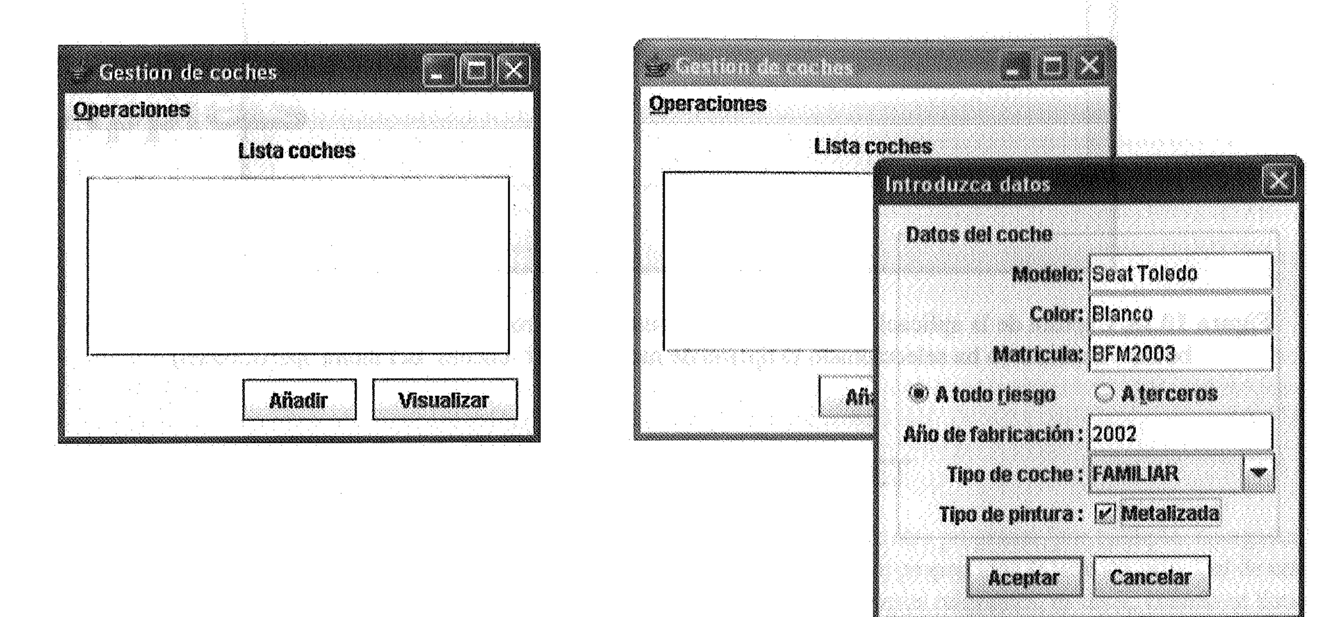


Figura 10.26. Capturas de la ventana inicial de la aplicación y con el cuadro de diálogo emergente que aparece al hacer clic en el botón `Añadir`. En este cuadro de diálogo se reutiliza el panel de datos del coche desarrollado en el Ejemplo 10.15 para introducir los datos del coche.

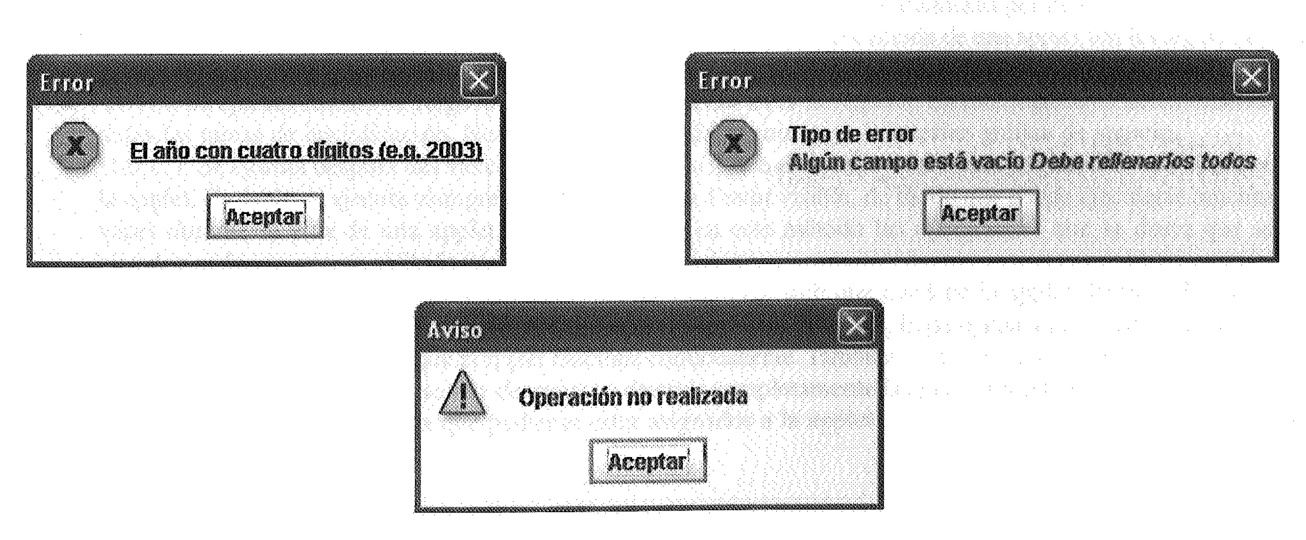


Figura 10.27. Capturas de los cuadros de diálogo que indican que se han producido errores en la entrada de datos o que se ha pulsado el botón `Cancelar` del cuadro de petición de datos. Los dos cuadros de error utilizan las capacidades de texto HTML para mejorar la presentación del mensaje de error.

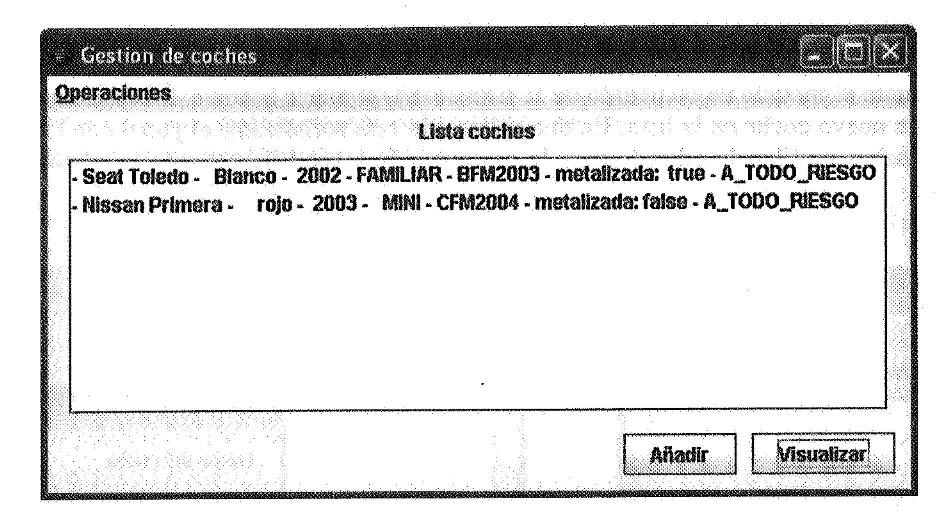
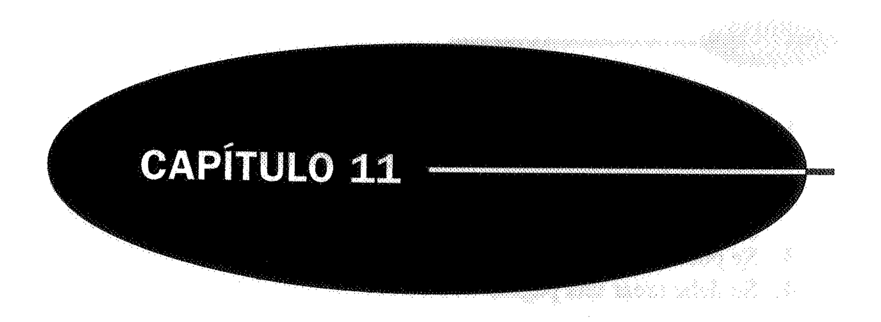


Figura 10.28. Captura de la aplicación una vez que el usuario ha introducido dos coches y ha hecho clic en el botón Visualizar (o ha seleccionado la opción de menú `Listar Coches` del menú `Operaciones`).



Applets

11.1 ENTORNO Y CICLO DE VIDA DE UNA APPLET

En realidad una applet es un tipo especializado de panel que se ejecuta dentro de otro programa (por ejemplo, navegador, AppletViewer) que le suministra la funcionalidad y el soporte que habitualmente proporciona la ventana principal de una aplicación. El programa dentro del que se ejecuta la applet le comunica los sucesos relevantes, es decir, cuándo se tiene que crear, cuándo se tiene que ejecutar, cuándo se tiene que detener y cuándo se tiene que destruir. Estos sucesos conforman el *ciclo de vida* de una applet. Como una applet se ejecuta en el entorno proporcionado por el navegador, normalmente no incluye ni método principal ni constructores. Su ejecución está determinada por unos métodos fijos de la applet que ejecuta directamente el navegador en respuesta a cada uno de los sucesos relevantes de su ciclo de vida. Por ejemplo, cuando se crea una applet se ejecuta el método `init()` y cuando se destruye se ejecuta el método `destroy()`. Las applets Swing se crean siempre especializando la clase `JApplet` que proporciona el comportamiento por defecto y el marco de trabajo para su creación. Los métodos que controlan la creación, ejecución y destrucción de una applet son los siguientes:

- `init()`. Se ejecuta cuando se carga la applet por primera vez en el navegador. En este método se deben realizar todas las tareas de inicialización. Normalmente es donde se construye la interfaz gráfica de usuario.
- `start()`. Se ejecuta después del método `init()` y cada vez que el usuario vuelve a la página web que contiene la applet. Es decir, se ejecuta siempre que la applet pasa a estar visible, de modo que puede ejecutarse muchas veces durante la vida de una applet. Deben incluirse en este método las operaciones que se desea que se ejecuten cada vez que se visite la página web que la contiene.
- `stop()`. Se ejecuta cada vez que el usuario abandona la página web que contiene la applet. Permite detener o suspender temporalmente operaciones costosas cuando la applet no es visible. Estas operaciones se deberían reanudar en el método `start()` para que la applet funcione correctamente. También se ejecuta antes de destruir la applet.
- `destroy()`. Se ejecuta cuando se va a descargar o destruir completamente la applet. En este método se deberían liberar los recursos del sistema que pudieran estar asignados a la applet.

11.2 CREACIÓN DE UNA APPLET

La creación de una applet es muy similar a la construcción de cualquier otra aplicación con interfaz gráfica de usuario. Los pasos habituales para la creación de una applet Swing son los siguientes:

1. La clase principal de la applet debe heredar de la clase JApplet que le proporciona la comunicación con el entorno y la funcionalidad básica de ejecución.
2. Se debe definir el método init() para inicializar todos los elementos de la applet. Aquí se construye la interfaz gráfica de usuario que en las aplicaciones gráficas normales se incluía en el constructor.
3. Se pueden definir los métodos start(), stop() y destroy() para obtener el comportamiento deseado.
4. Se debe crear una página web en HTML que contenga la applet.

Los aspectos de diseño y creación de la interfaz gráfica de usuario y de la gestión de eventos para controlar el funcionamiento de dicha interfaz se tratan de la misma forma que con las aplicaciones gráficas presentadas en el Capítulo 10.

11.3 CLASES APPLET Y JAPPLET

Las principales clases implicadas en la creación de las applets Swing son java.applet.Applet y javax.swing.JApplet.

El paquete java.applet proporciona parte de las clases e interfaces necesarias para crear una applet Swing. En este paquete se define la clase Applet que proporciona el comportamiento y funcionalidad básica (por ejemplo, los métodos para responder a los sucesos del ciclo de vida de una applet) y otros métodos para que la applet pueda comunicarse con su entorno de ejecución. Las applets también incorporan funcionalidades multimedia que permiten incorporar sonidos e imágenes de una forma sencilla.

La clase principal para la creación de applets Swing es JApplet, incluida en el paquete básico de Swing javax.swing. Es una especialización de Applet que proporciona todas las funcionalidades de un contenedor de alto nivel. Por tanto, las applets Swing tienen un panel de contenido y la capacidad de incorporar una barra de menú. El administrador de disposición por defecto de JApplet es BorderLayout.

11.4 HTML, XHTML Y LAS APPLETS: LA MARCA <APPLET> Y LA MARCA <OBJECT>

Una applet debe tener asociada una página web que, cuando se carga en un navegador de Internet, provoca su ejecución. Las páginas web se construyen utilizando un lenguaje de marcado denominado lenguaje de marcado de hipertexto (HTML, *HyperText Markup Language*) que ha pasado a preferirse XHTML más recientemente, ya que el lenguaje de marcado incluye, por ejemplo, marcas tanto de inicio como de finalización.

Una página web es un archivo de texto creado con HTML en el que se puede diferenciar dos tipos de contenidos: los contenidos de información propiamente dichos y las marcas que identifican, delimitan y organizan dichos contenidos.

Además de las marcas de presentación, existe una marca especial <applet> que indica que hay una applet asociada a la página y especifica cuál es la clase que se debe ejecutar cuando se visualiza la página. Mediante atributos especificados dentro de la marca <applet>, además de la clase a ejecutar, se puede proporcionar información complementaria como, por ejemplo, el directorio en el que se encuentra dicha clase, el tamaño y posición de la applet, los argumentos que se le proporcionan para la ejecución, e incluso un texto alternativo en caso de que el navegador no sea compatible Java y no se pueda ejecutar correctamente la applet. A continuación se presenta un ejemplo de página Web para ver una applet llamada CicloVidaApplet.class, donde la applet se desea presentar en un panel de 200 por 200 con color de fondo azul claro.

```
<html>
<head>
     <title>CicloVidaApplet</title>
     <style type="text/css">
        body {background-color: lightblue} </style>
</head>
```

```
<body>
    <h1> El applet demostrativo del ciclo de vida </h1>
    <APPLET CODE = "CicloVidaApplet.class"
    WIDTH = 200 HEIGHT = 200></APPLET>
</body>
</html>
```

A una applet también se le pueden proporcionar datos o argumentos para su ejecución desde la página web mediante la etiqueta `<param>` que permite especificar parámetros mediante su nombre y su valor. La applet puede acceder a dichos datos mediante el método `getParameter()`. Si en el ejemplo previo se hubiera incluido en la línea anterior al mensaje alternativo `<param name=nombre value="Celia">`, el resultado de haber ejecutado `getParameter("nombre")` en alguno de los métodos de la applet habría sido `Celia`. Una applet puede tener más de un parámetro.

La marca APPLET plantea el problema de que sólo funciona para las applets de Java por lo que en HTML 4 y posteriormente en XHTML se optó por utilizar una solución genérica para la inclusión de objetos (o programas) que es la marca OBJECT y, por tanto, se desaconsejó el uso de APPLET. No obstante, por lo breve de esta introducción y por que existen problemas de incompatibilidad con algunos navegadores, hemos preferido continuar con APPLET. Para obviar todos los problemas planteados con el uso de la nueva marca OBJECT y con las posibles diferencias que se puedan generar por el comportamiento de los diferentes navegadores en la distribución estándar de Java, se incluye una utilidad denominada `HtmlConverter` que se puede encontrar en el subdirectorio `bin` del directorio de instalación de Java.

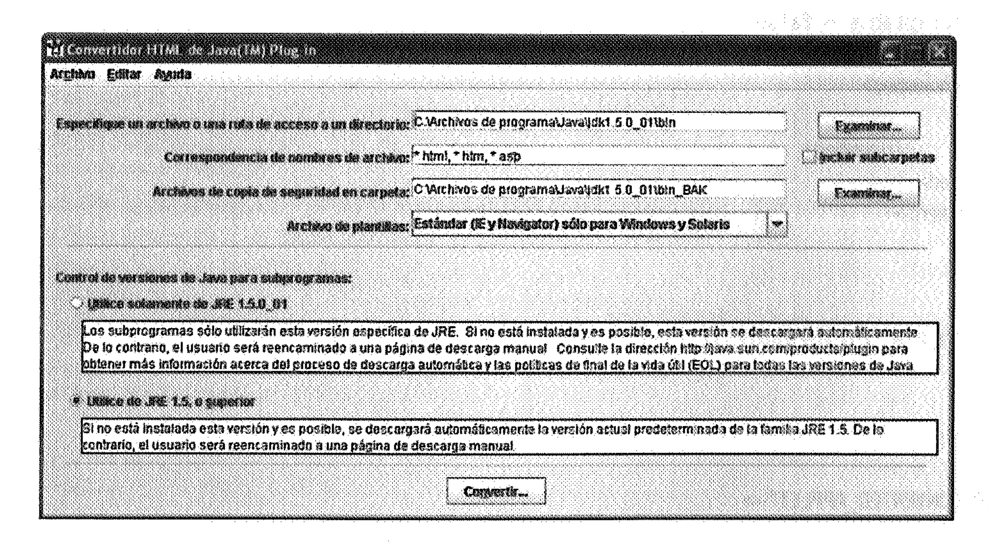


Figura 11.1. Captura de la pantalla del convertidor HTML de Java (`HtmlConverter`).

El resultado de aplicar este convertidor de HTML sobre el fichero HTML con la marca APPLET correspondiente al ejemplo anterior del ciclo de vida es el siguiente:

```
<html>
<head>
    <title>CicloVidaApplet</title>
    <style type="text/css">
       body {background-color: lightblue} </style>
```

```
</head>
<body>
    <h1> El applet demostrativo del ciclo de vida </h1>

    <!-"CONVERTED_APPLET"->
<!- HTML CONVERTER ->

<object
    classid = "clsid:8AD9C840-044E-11D1-B3E9-00805F499D93"
    codebase =
"http://java.sun.com/update/1.5.0/jinstall-1_5-windows-i586.cab#Version=1,5,0,0"
    WIDTH = 200 HEIGHT = 200 >
    <PARAM NAME = CODE VALUE = "CicloVidaApplet.class" >
    <param name = "type" value = "application/x-java-applet;version=1.5">
    <param name = "scriptable" value = "false">

    <comment>
    <embed
            type = "application/x-java-applet;version=1.5" \
            CODE = "CicloVidaApplet.class" \
            WIDTH = 200 \
            HEIGHT = 200
        scriptable = false
        pluginspage = "http://java.sun.com/products/plugin/index.html#download">
        <noembed>

            </noembed>
    </embed>
    </comment>
</object>

<!-
<APPLET CODE = "CicloVidaApplet.class" WIDTH = 200 HEIGHT = 200>


</APPLET>
->
<!-"END_CONVERTED_APPLET"->

</body>
</html>
```

No obstante, hay que tener cierto cuidado con las applets ya que en función de la configuración e instalación del sistema puede provocar la descarga de una gran cantidad de datos remotos con el consiguiente retraso. Además, las versiones más actualizadas de los navegadores (e incluso algunos complementos para los navegadores o programas antivirus) pueden limitar o prohibir la ejecución de las applets. En cualquier caso, siempre es posible hacer las pruebas ya que Sun en su distribución estándar incluye un visualizador de applets denominado AppletViewer.

Para obtener más información sobre el programa de conversión de HTML a las nuevas recomendaciones, consulte el sitio web de Sun (*www.sun.com*). Se puede encontrar más información sobre XHTM en el sitio web del World Wide Web Consortium (*www.w3c.org*).

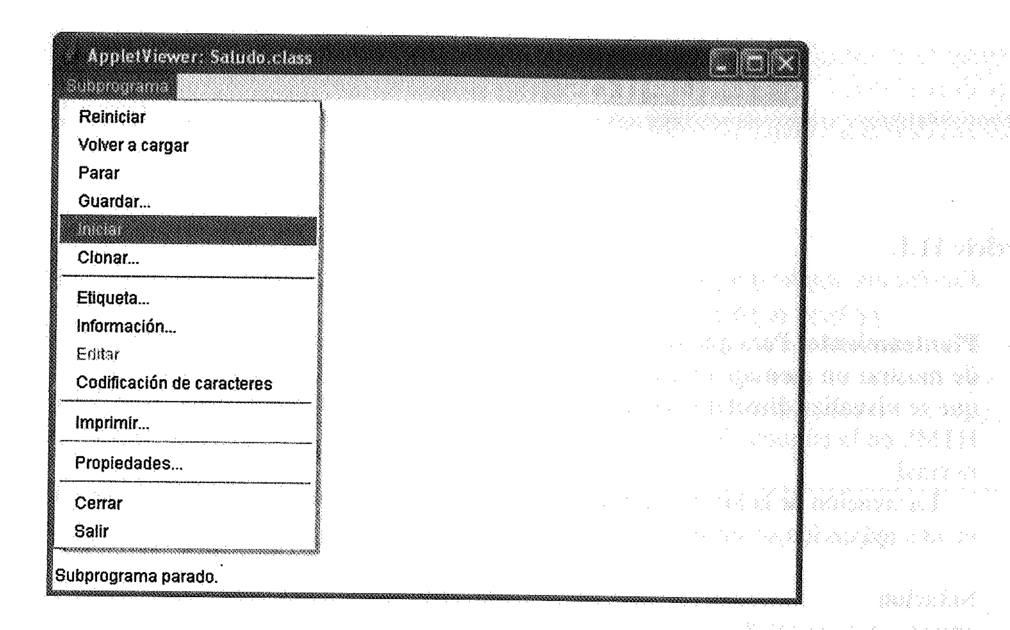


Figura 11.2. Captura de la pantalla del visualizador de applets de Java (`AppletViewer`). Desde su menú se pueden reproducir los eventos del ciclo de vida y acceder a algunas de las características de la applet.



Problemas resueltos

APPLETS

Ejercicio 11.1:

Escriba una applet que presente un mensaje de saludo en la pantalla.

Planteamiento: Para que sea una applet se crea una clase que herede de JApplet. La forma más sencilla de mostrar un mensaje es crear una etiqueta con el mensaje y añadir dicha etiqueta a la applet de forma que se visualiza directamente. Para mejorar la presentación del mensaje se utilizan las características del HTML en la etiqueta de modo que el texto aparece destacado y con un tamaño de fuente que es mayor del normal.

La creación de la interfaz gráfica en una applet habitualmente se hace en el método init(). Recuerde que en una aplicación se suele crear la interfaz en el constructor.

Solución:

```java
import javax.swing.*;

public class AppletSaludoEtiqueta extends JApplet {

    public void init() {
        String textoHtml ="<html><strong><font size=+3>Un saludo a todos";
        JLabel etiqueta = new JLabel(textoHtml, JLabel.CENTER);
        add(etiqueta);
    }

}//AppletSaludoEtiqueta
```

Fichero html:

```html
<html>
<head>
     <title>Primera applet de saludo</title>
     <style type="text/css">
        body {background-color: lightblue} </style>
</head>
<body>
<center>
<applet
    code    = "AppletSaludoEtiqueta.class"
    width   = "500"
    height  = "300"
    >
</applet>
</center>
</body>
</html>
```

Comentario: Fíjese que para que el mensaje aparezca centrado en el panel de la applet se ha cambiado la alineación horizontal de la etiqueta a JLabel.CENTER.

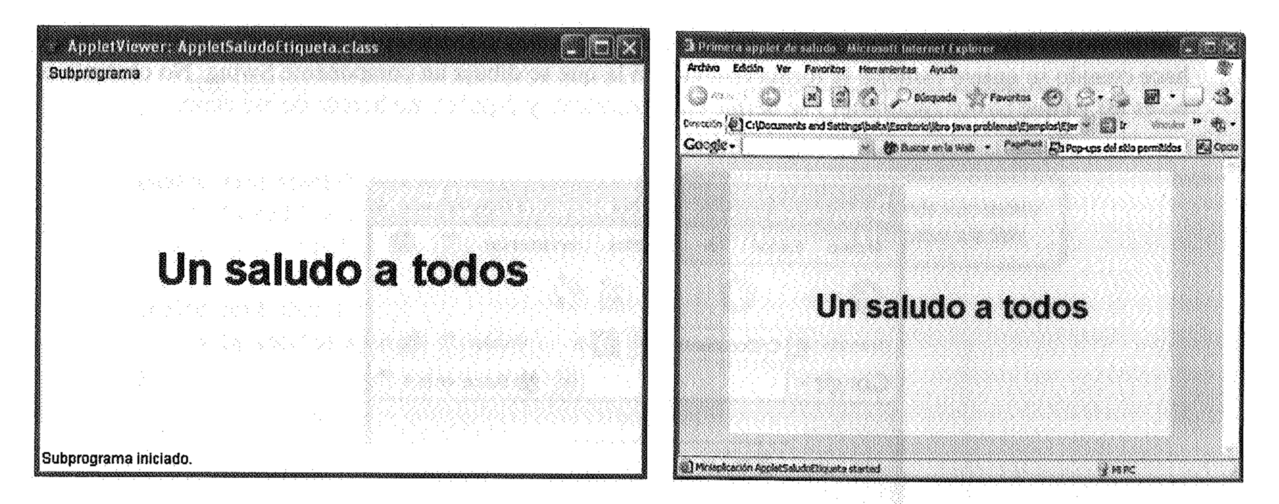


Figura 11.3. Captura de la ejecución de la applet en el AppletViewer y en el navegador.

Ejercicio 11.2:

Escriba una applet que presente un mensaje de saludo en la pantalla.

Planteamiento: Otra forma habitual de escribir en las applets es modificando la forma en la que se presenta o dibuja la propia applet. Para hacerlo hay que reescribir el método `paint(Graphics g)`. En concreto, para este ejercicio se modifica su forma de presentación estándar dibujando en su área de visualización dos líneas de color azul (mediante `drawLine()`) que enmarcan el saludo que aparece en color rojo. La escritura debe hacerse en modo gráfico por lo que hay que proporcionar las coordenadas donde se quiere que aparezca.

La especificación del tamaño que ocupará la applet se hace en el archivo HTML asociado. No obstante se puede producir un conflicto, como en este caso, que en el código se decide redimensionar la applet. Habitualmente los navegadores aplican el tamaño definido en el archivo HTML.

Solución:

```
import java.awt.*;
import javax.swing.*;

public class Saludo extends JApplet {

    public void init() {
        resize(160, 120);
    }

    public void paint(Graphics g) {
        super.paint(g); //pintado del fondo
        g.setColor(Color.BLUE);
        g.drawLine(10, 10, 150, 10);
        g.setColor(Color.RED);
        g.setFont(new Font("Courier", Font.BOLD, 20));
        g.drawString("Hola Mundo", 20, 30 );
        g.setColor(Color.BLUE);
        g.drawLine(10, 45, 150, 45);
    }
}//Saludo
```

Se cambia el tamaño de la applet.

Se fija la fuente, el estilo y el tamaño en el que se escribe.

Comentario: Fíjese que podría parecer que el método que hay que reescribir es `paintComponent()` como se hace cuando se quiere cambiar la forma estándar en la que se dibuja un componente Swing. No obstante esto no es así ya que `paintComponent()` se define en `JComponent` y `JApplet` no hereda de esa clase.

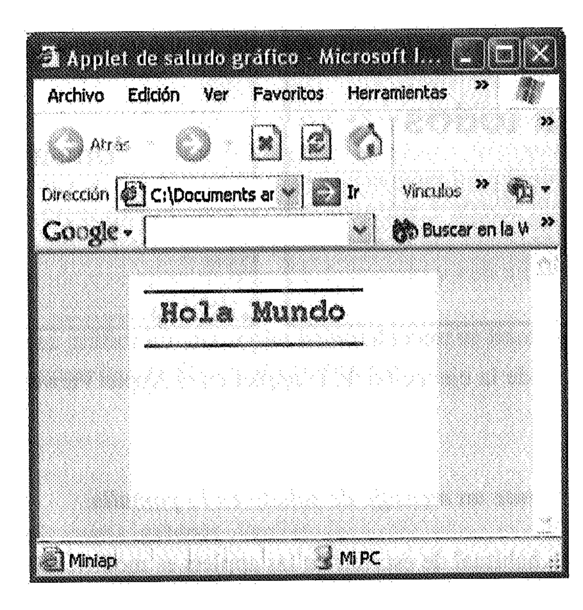


Figura 11.4. Captura de la ejecución de la applet de saludo en el navegador.

Ejercicio 11.3:

Escriba una applet que muestre el ciclo de vida de estas aplicaciones dentro del navegador. Es decir que presente en la pantalla una cadena de texto explicativa con cada uno de los sucesos del ciclo de vida de una applet.

Planteamiento: Para crear la ventana, como las applets tienen asignado por defecto el gestor de diseño `BorderLayout`, se usa la parte superior para añadir una etiqueta que mejore la legibilidad y en la parte central se coloca un panel que refleje los sucesos que le van pasando a la applet. Este no será un panel estándar si no uno particularizado que muestre los sucesos directamente cuando se visualice. Como hay que mostrar todos los sucesos y no sólo el último, hay que almacenar dichos sucesos. Para ello, se declara una lista como atributo de la applet.

Solución:

```
import java.awt.*;
import javax.swing.*;
import java.util.*;

public class CicloVidaApplet extends JApplet
{
    private java.util.List<String> lista;

    public void init() {
        lista = new ArrayList<String>();
        lista.add("Inicialización en init()");
        add(new JLabel("Applet: ciclo de vida"), BorderLayout.PAGE_START);
```

La lista de cadenas se instancia con un `ArrayList`.

```
        MiPanel panel = new MiPanel();
        add(panel);
    }

    public void start() {
        lista.add("se ejecuta start()");
    }

    public void stop() {
        lista.add("se ejecuta stop()");
    }

    public void destroy() {
        lista.add("se ejecuta destroy()");
    }

    class MiPanel extends JPanel {
        MiPanel() {
            super();
        }
        public void paintComponent(Graphics g) {
            super.paintComponent(g);
            for (int i=10, cont=0 ; cont <lista.size(); cont++, i=i+15)
                g.drawString(lista.get(cont), 15,i );
        }
    } // MiPanel
}//CicloVidaApplet
```

Creación de la interfaz de usuario.

Cada método que responde a un suceso añade una cadena explicativa a la lista.

Dibujado del fondo del panel.

Escritura de forma gráfica de las cadenas explicativas de los sucesos.

AppletViewer: ...
Subprograma
Applet: ciclo de vida
Inicialización en init()
se ejecuta start()
Subprograma iniciado.

AppletViewer: ...
Subprograma
Applet: ciclo de vida
Inicialización en init()
se ejecuta start()
se ejecuta stop()
se ejecuta start()
Subprograma iniciado.

Figura 11.5. Captura de la applet ejecutándose en el AppletViewer. A la izquierda se presenta la captura inicial y a la derecha después de haber minimizado y vuelto a su tamaño normal la ventana.

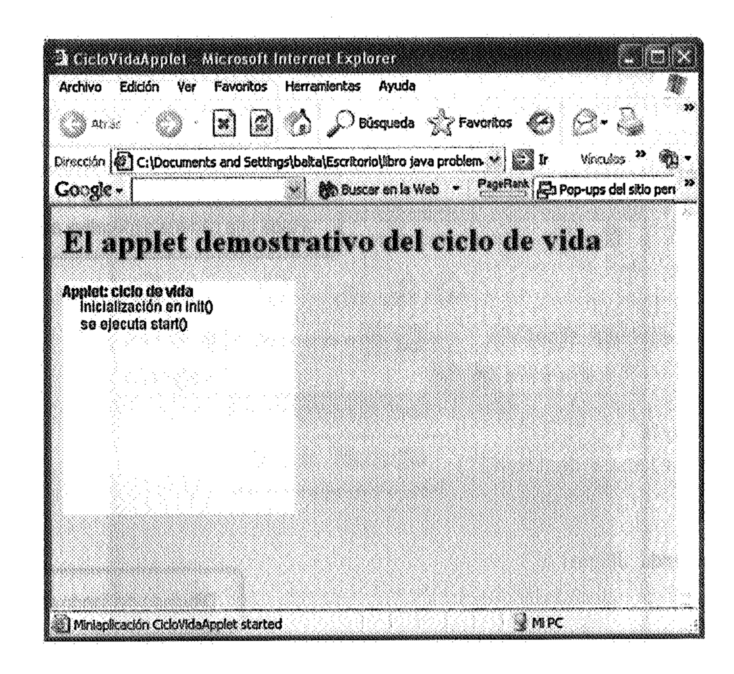


Figura 11.6. Captura de la applet ejecutándose en el navegador Microsoft Internet Explorer.

Comentario: Fíjese en que como la declaración de lista para almacenar los sucesos del ciclo de vida se hace mediante la interfaz lista, posteriormente se instancia mediante un `ArrayList` pero podría haberse instanciado con cualquier otra clase que implemente dicha interfaz (por ejemplo, `LinkedList`).

La clase `MiPanel` especializa a la clase `JPanel` y en ella se ha redefinido el método `paintComponent()`(y no `paint()`como se ha hecho en el ejemplo anterior en la applet de saludo gráfico) para que al mostrarse presente en su área de visualización el contenido de la lista de cadenas. Por eso para la escritura de las cadenas se utiliza el método de escritura gráfica `drawString()`, donde además de proporcionar la cadena a mostrar hay que proporcionar las coordenadas donde se desea mostrar dicha cadena dentro del componente.

Ejercicio 11.4:

Escriba una applet que también pueda ejecutarse como una aplicación normal.

Planteamiento: Una applet es una especialización de un panel AWT que no genera directamente una ventana principal en la cual mostrarse. Por tanto para que pueda ejecutarse como una aplicación hay que añadir un programa principal donde se genere esa ventana principal de la aplicación (`JFrame`) y una instancia de la applet que se añade a dicha ventana. Posteriormente, para que se ejecute como aplicación, se debe invocar expresamente por lo menos al método `init()` (y habitualmente también al método `start()`) ya que ahora no está incluido en un entorno de ejecución más general que le proporcione dichos eventos.

Para este ejemplo hemos reutilizado y ampliado la applet del Ejemplo 11.1.

Solución:

```
import javax.swing.*;

public class extends JApplet {

    public void init() {
        String textoHtml = "<html><strong><font size=+3>Aplicacion y Applet";
```

```
        JLabel etiqueta  = new JLabel(textoHtml, JLabel.CENTER);
        add(etiqueta);
    }

    public static void main( String[] args ) {
        JFrame ventana = new JFrame("Ventana de la aplicación");
        AppletAplicacion applet = new AppletAplicacion();
        ventana.add(applet);
        applet.init();
        applet.start();
        ventana.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        ventana.setSize(500, 300);
        ventana.setVisible(true);
    }

}// AppletAplicacion
```

Se crea la applet y se añade a la ventana.

Comentario: Fíjese que, por generalidad, en el `main()` se invoca al método `start()` de la applet aunque no se haya escrito. Esto no provoca ningún tipo de error de ejecución.

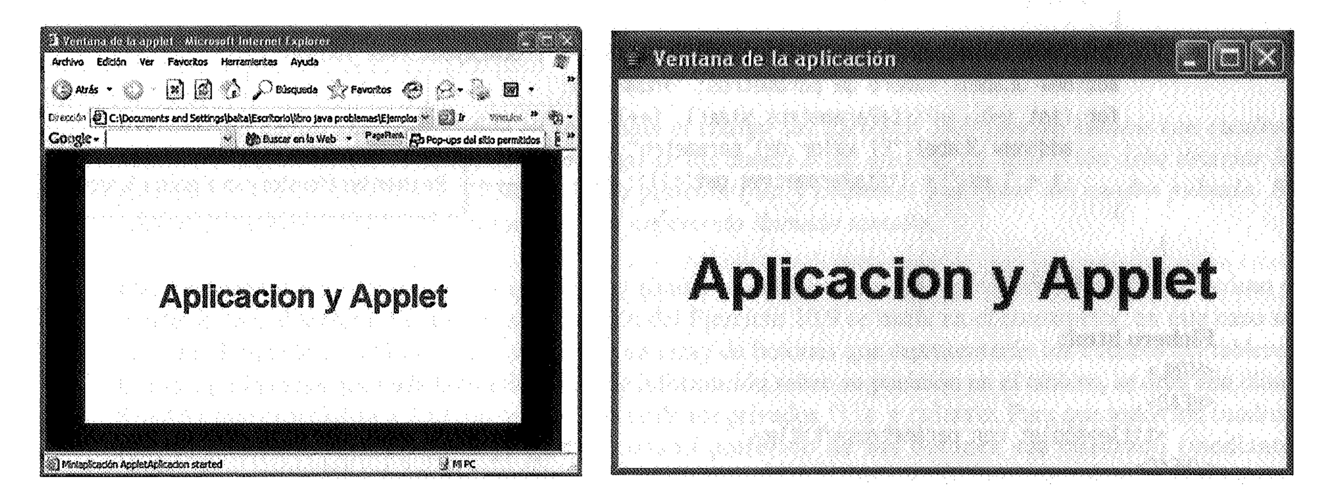


Figura 11.7. Captura de la ejecución, primero como applet en el navegador y luego como aplicación normal.

Ejercicio 11.5:

Escriba una applet que acepte un número variable de parámetros y muestre su valor por pantalla. El número variable de parámetros de cada ejecución vendrá dado por el valor de un parámetro cuyo nombre es numeroParametros. El resto de los parámetros tendrán nombres parámetro# (donde # representa el número desde 1 hasta el valor de numeroParametros)

Planteamiento: En este caso utilizamos el método de añadir etiquetas para escribir tanto el número de parámetros que se indican en el archivo HTML asociado al applet como sus valores. Por tanto primero se obtiene el valor del número de parámetros y luego se obtienen los valores de dichos parámetros que se almacenan en una lista de cadenas. Esta lista de cadenas se implementa mediante un `ArrayList`.

Una vez que se han obtenido todos los valores de los parámetros sólo queda crear las etiquetas que tengan por texto dicho valor y añadirlas a la applet.

Solución:

```
import javax.swing.*;
import java.util.*;

public class AppletConParametros extends JApplet {
    java.util.List<String> listaParametros;
    public void init() {
        listaParametros = new ArrayList<String>();
        int numPar = 0;
        String nombreParametro = "numeroParametros";
        String valor = this.getParameter(nombreParametro); ← Se obtiene el número de parámetros.
        try{
            numPar = Integer.parseInt(valor);
        }catch (NumberFormatException e){
            System.out.println("El valor del número de parámetros no es un número");
        }
        for (int i=1; i<=numPar; i++){
            nombreParametro = "parametro"+i;;
            valor = this.getParameter(nombreParametro); ← Se obtienen los valores de los parámetros y se añaden a la lista.
            listaParametros.add(valor);
        }
        this.setLayout(new BoxLayout(this.getContentPane(), BoxLayout.PAGE_AXIS));
        add(new JLabel("Numero de parámetros: "+numPar));
        for (int i=0; i<listaParametros.size(); i++){
            add(new JLabel("El valor del parametro" +
             i + " es " + listaParametros.get(i))); ← Se crean las etiquetas y se añaden a la applet.
        }
    }
}//AppletConParametros
```

Fichero html:

```
<HTML>
<HEAD>
    <title>Applet con parámetros</title>
</HEAD>

<BODY BGCOLOR="blue">
<CENTER>
<APPLET code= "AppletConParametros.class" width = "500" height  = "300">
    <param name=numeroParametros value="4">
    <param name=parametro1 value="Julian">
    <param name=parametro2 value="Antonio">
    <param name=parametro3 value="Francisco">
    <param name=parametro4 value="Basilio">
</APPLET>
</CENTER>
</BODY>
</HTML>
```

Comentario: Fíjese que el valor de los parámetros se obtiene a partir del nombre del parámetro y no por el orden de aparición, que no tiene ninguna importancia.

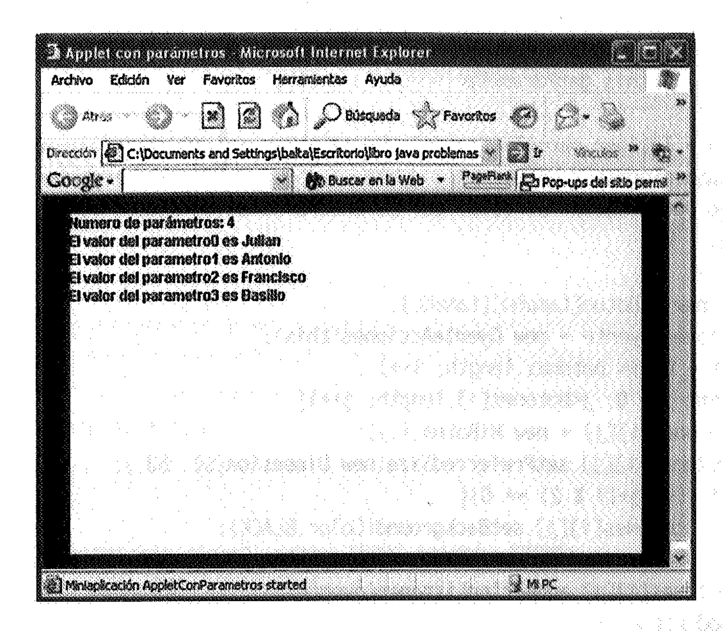


Figura 11.8. Captura de la applet en el navegador Microsoft Internet Explorer.

Ejercicio 11.6:

Escriba una applet que reutilizando en lo posible el código del Ejercicio 10.9 cree un tablero con cuadros blancos y negros similar al utilizado en el juego de las damas o del ajedrez. La aplicación debe detectar la pulsación sobre el tablero e informar sobre la posición (fila y columna) y el color del cuadro pulsado. El tamaño del tablero estará determinado por un argumento llamado tamaño.

Planteamiento: En este caso el código es muy similar al del Ejercicio 10.9 pero hay diferencias en cómo y dónde se crea el tablero. Lo que en la aplicación del Ejercicio 10.9 se hacía en el constructor, en este caso se hace en el método `init()` en el que se declara un array de botones que representarán los cuadros del tablero. Como aquí interesa que cada botón disponga de información sobre su posición en el tablero, se crea una clase `MiBoton` que especializa a `JButton` y tiene dos atributos privados `fila` y `columna`. Para que todos los cuadros tengan el mismo tamaño se elige que el tamaño preferido de los botones sea cuadrado (mediante `setPreferredSize()`). Para representar los cuadros negros se establece su color de fondo a dicho color (`setBackground(Color.BLACK)`).

Por defecto, el gestor de disposición o administrador de diseño del panel es el `BorderLayout` que organiza su contenido por regiones. Como interesa organizar los cuadros en forma de tablero se cambia el gestor de disposición por defecto por el gestor `GridLayout(filas, columnas)` que organiza los componentes en una matriz de tamaño dado por las filas y las columnas. En este caso el tamaño del tablero se obtiene del parámetro tamaño del archivo HTML.

Solución:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class TableroBotones extends JApplet {
    public void init() {
        MiBoton[][] botones;
        int tamaño = 0;
```

```
    String nombreParametro = "tamaño";
    String valor = this.getParameter(nombreParametro);
    try{
        tamaño = Integer.parseInt(valor);  <--- Se obtiene el tamaño del tablero.
    }catch (NumberFormatException e){
        System.out.println("El valor del parámetro no es un número");
        tamaño = 9;
    }
    botones = new MiBoton[tamaño][tamaño];
    OyenteAcciones oyente = new OyenteAcciones(this);
    for (int i = 0; i< botones.length; i++)
        for (int j = 0; j<botones[i].length; j++){
            botones[i][j] = new MiBoton(i,j);
            botones[i][j].setPreferredSize(new Dimension(50, 50));
            if (((i+j+1) % 2) == 0){
                botones[i][j].setBackground(Color.BLACK);
            }
            botones[i][j].addActionListener(oyente);
            add(botones[i][j]);
    }
    setLayout(new GridLayout(tamaño,tamaño));
}//init

class MiBoton extends JButton {
    private int fila;  <--- Se declara la clase de botones MiBoton con fila y columna.
    private int columna;

    public MiBoton(int fila, int columna){
        this.fila    = fila;
        this.columna = columna;
    }

    public int getFila(){
        return fila;
    }

    public int getColumna(){
        return columna;
    }
}//MiBoton

class OyenteAcciones implements ActionListener{
    private JApplet applet;
    public OyenteAcciones(JApplet applet){
        this.applet = applet;
    }
    public void actionPerformed (ActionEvent evento){
        MiBoton boton = (MiBoton)evento.getSource();
        String color  = "blanco";
        if (boton.getBackground()== Color.BLACK)
            color = "negro";
```

```
                JOptionPane.showMessageDialog(applet,
                        "Se ha pulsado el cuadro (" + boton.getFila() + ", " +
                        boton.getColumna() + ")  de color " + color,
                        "Cuadro pulsado",
                        JOptionPane.INFORMATION_MESSAGE);
            }
        } //OyenteAcciones
    }//TableroBotones
```

Se obtiene la fila y la columna del botón pulsado.

Fichero html:

```
<HTML>
<HEAD>
   <title>Tablero en applet</title>
</HEAD>
<BODY BGCOLOR='blue'>
<CENTER>
<APPLET   code= "TableroBotones.class" width= "500" height= "300">
    <param name=tamaño value="8">
</APPLET>
</CENTER>
</BODY>
</HTML>
```

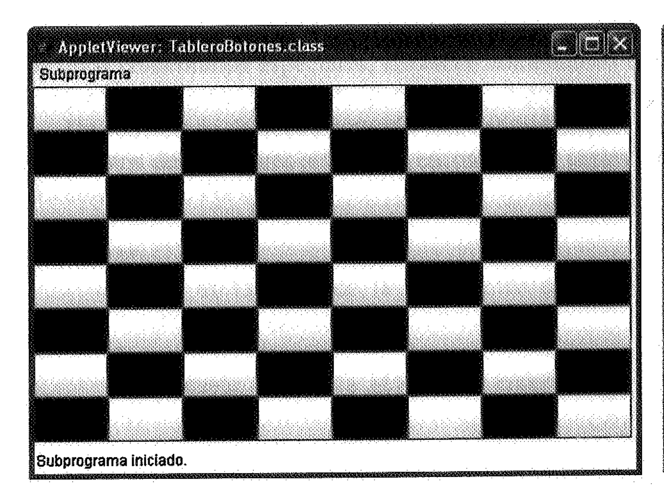
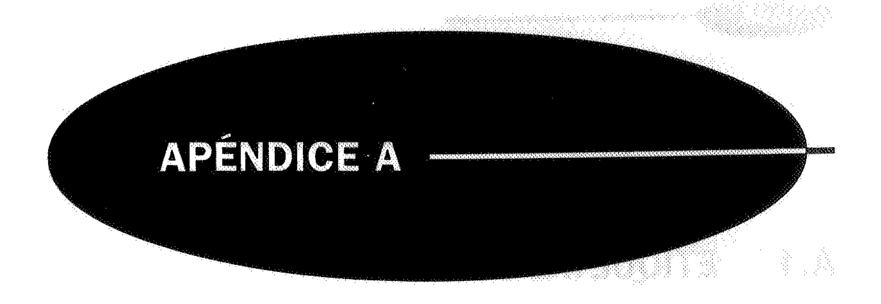




Figura 11.9. Captura de la applet inicial y después de haber pulsado el cuadro de la posición (4, 4). Fíjese que las ventanas emergentes indican que se han generado desde una applet.



Documentación del código

En Java la documentación del código es fundamental. Resulta evidente en cuanto se consulta la documentación de las clases que proporciona directamente en los paquetes que distribuye Sun. Toda la documentación se ha generado utilizando una herramienta, llamada javadoc. También podrá hacer lo mismo con todos sus programas.

Java promueve que el programador documente el funcionamiento de las clases dentro del propio código. A partir de unos comentarios especiales, llamados comentarios de documentación, con la herramienta javadoc se genera un conjunto de páginas en HTML por las que se puede navegar utilizando cualquier navegador Web.

Un comentario de documentación está delimitado por una marca de comienzo (/**) y una marca de terminación (*/). Entre estos delimitadores se pueden encontrar dos partes, una de descripción y otra parte con cero o más etiquetas de documentación. Por ejemplo:

```
/**
 * Esta es la parte de la descripción donde se
 * comenta lo que venga a continuación
 *
 * @etiqueta Uso de la etiqueta y comentario
 */
```

En este ejemplo se puede ver la estructura normal que uno se puede encontrar en un comentario de documentación. Éste empieza con el delimitador de comentario de documentación (/**) en una línea.

El resto del comentario empieza siempre cada línea con un carácter asterisco. Estos asteriscos al principio de la línea desaparecen cuando se generan las páginas HTML a partir de esta documentación. Los entornos de programación suelen ayudar a generar este tipo de comentarios especiales de forma que el programador no tenga que preocuparse por dar este formato especial.

La última línea termina con el delimitador de fin de comentario de documentación (*/).

Un comentario de documentación tiene dos partes bien diferenciadas:

- En la primera parte va un texto en el que el programador escribe el comentario sobre la clase, atributo, constructor o método que venga a continuación; puede poner todo el texto que desee. Además, puede incluir etiquetas propias del lenguaje HTML que se mantendrán en la versión final, de manera que las interprete posteriormente el navegador Web. De esta forma se puede dar cierto formato al texto que aparecerá en la página Web de documentación.

- En la segunda parte se pone la lista de etiquetas de documentación con cierto texto que aparecerá después en apartados específicos de la documentación. Estas etiquetas tienen un significado especial que permite darles un formato propio en la documentación para destacar dicha información.

A.1 ETIQUETAS Y POSICIÓN

No todas las etiquetas se pueden utilizar en cualquier comentario de documentación. Hay etiquetas que no tienen sentido delante de un constructor de una clase, como por ejemplo @param, y tampoco para describir un atributo. Existen dos tipos de etiquetas de documentación:

- Etiquetas de bloque: Son las etiquetas que sólo se pueden incluir en el bloque de documentación (ver sección anterior). Estas etiquetas son las que en la tabla comienzan por @.
- Etiquetas en el texto: Son las etiquetas que se pueden poner en cualquier punto de la descripción o en cualquier punto de la documentación asociada a una etiqueta de bloque. Estas etiquetas son las que en la tabla están definidas entre llaves como {@code}

En la Tabla A.1 puede ver la lista completa de etiquetas, el ámbito en el que se puede emplear dicha etiqueta y la versión de Java en la que aparece por primera vez.

Tabla A.1. Lista de etiquetas de documentación y ámbito de uso.

	@author	{@code}	{@docRoot}	@deprecated	@exception	{@inheritDoc}	{@link}	{@linkplain}	{@literal}	@param	@return	@see	@serial
Descripción	X		X	X			X	X				X	
Paquete	X		X	X			X	X				X	
Clases e interfaces	X		X	X			X	X				X	X
Atributos			X	X			X	X				X	X
Constructores y métodos			X	X	X	X	X	X		X	X	X	
Aparece en el JDK/SDK	1.0	1.5	1.3	1.0	1.0	1.4	1.2	1.4	1.5	1.0	1.0	1.0	1.2

A.2 USO DE LAS ETIQUETAS

En esta sección se describe solamente el uso de las etiquetas de documentación más habituales. Si desea una descripción completa consulte la documentación de Java en *java.sun.com*.

@author *texto con el nombre*
El texto después de la etiqueta no tiene que tener ningún formato especial. Se pueden incluir múltiples etiquetas @author una detrás de otra o poner varios nombres de autores en la misma etiqueta.

```
@author Jesús Sánchez Allende
```

@version *texto de la versión*
El texto de la versión no tiene ningún formato especial. Se recomienda poner el número de la versión y la fecha de la misma. Se pueden incluir múltiples etiquetas @version una detrás de otra o poner varios nombres de autores en la misma etiqueta.

```
@version 1.2 29-Enero-2004
```

@deprecated *texto*
Indica que no debería de utilizarse, indicando en el texto las causas de ello. Se puede utilizar en todos los apartados de documentación. Si se ha realizado una sustitución debería indicarse qué utilizar en su lugar.

```
@deprecated El método no actúa correctamente. Utilice en su lugar {@link metodoCorrecto}.
```

@exception *nombre-excepción texto*
Esta etiqueta es un sinónimo de @throws

@param *nombre-atributo texto*
Al atributo (parámetro) le debe seguir el nombre del atributo que se va a comentar. Resulta de interés indicar en el comentario el tipo o clase del argumento, así como su uso y límites de valores, si existen. Como texto puede escribir tantas líneas como sean necesarias.

```
@param numAlumnos El número de alumnos en el grupo
```

@return *texto*
Se puede omitir en los métodos que devuelven void. Debe de aparecer en todos los métodos, dejando explícito qué tipo o clase de valor devuelve y sus posibles rangos de valores.

```
@return Devuelve el índice del elemento buscado. -1 si no existe.
```

@see *referencia*
Añade una sección "See Also" con un enlace y un comentario sobre la referencia. Existen tres tipos básicos:

- @see "texto" – Aparece el texto tal y como se ha escrito entre comillas. Sirve para hacer referencia a libros, artículos, etc.
- @see texto - añade el enlace dado.
- @see paquete.clase#miembro texto – Añade una referencia a un elemento del programa.

Si existe sobrecarga de métodos hay que indicar el tipo de los argumentos para que el generador pueda interpretar correctamente a cuál de ellos se refiere. En el caso de que el argumento sea de una clase, se debe especificar el paquete completo. Si al final se añade un nombre, aparece dicho nombre como enlace de hipertexto hacia el elemento marcado.

```
@see #metodo El método indicado
@see java.lang.Integer
@see Integer
```

```
@see Integer#MAX_VALUE El mayor valor posible
@see Integer#toString(int)
@see Integer#valueOf(java.lang.String)
```

@throws *nombre-excepción texto*
El nombre de la excepción es su nombre completo. Añada uno por cada excepción que se lance explícitamente con una cláusula throws, siguiendo un orden alfabético. En Java 2 se puede utilizar la etiqueta @throws como sinónimo de @exception.

```
@throws numeroNegativoException El valor del argumento no puede ser negativo.
```

@since *texto*
Sirve para indicar a partir de qué versión aparece esta característica o elemento en el paquete donde se encuentra.

```
@since 1.5
```

@version *texto*
Sirve para indicar cuál es la versión actual del elemento. Pueden utilizarse varios.

```
@version 2.4 03/03/2005
```

A.3 ORDEN DE LAS ETIQUETAS

El orden de las etiquetas debe ser el siguiente:

- `@author`: En clases e interfaces. Se pueden poner varios. En este caso resulta apropiado hacerlo en orden cronológico.
- `@version`: En clases y en interfaces.
- `@param`: En métodos y constructores. Se ponen tantos como argumentos tenga el constructor o el método. Deberían aparecer en el mismo orden en que se declaran.
- `@return`: En métodos.
- `@exception`: En constructores y métodos. Deberían aparecer en el mismo orden en que se declaran o en orden alfabético.
- `@throws`: Con Javadoc 1.2 es un sinónimo de @exception.
- `@see`: Se pueden poner varios. Se recomienda empezar por los más generales e ir indicando después los más concretos.
- `@since`.
- `@deprecated`.

A.4 EJEMPLO DE DOCUMENTACIÓN DE UNA CLASE

En esta sección se pone un ejemplo de cómo se puede comentar una clase siguiendo las recomendaciones de documentación anteriores.

```
/*
 * Clase Grupo.java
 * Java 2. Libro de ejercicios de Java. Documentación de código
 * Año 2005
 *
```

```
* Jesús Sánchez Allende
*/

import java.io.*;

/**
* La clase Grupo permite disponer de una reunión de
* alumnos de la clase Alumno. Todos los alumnos se
* guardan en un array cuyo tamaño se determina cuando
* se construye el objeto Grupo.
*
* @author  Jesús Sánchez Allende
* @author  Gabriel Huecas Fernández-Toribio
* @author  Baltasar Fernández Manjón
* @author  Pilar Moreno Díaz
* @version 2.2, Mayo de 2005
* @see     Alumno
*/
public class Grupo {
   /**
   * Nombre que permite identificar el grupo
   */
   private String nombre;

   /**
   * Array que contiene los alumnos del grupo.
   */
   private Alumno[] alumnos;

   /**
   * Variable que indica cuantos alumnos hay en el grupo.
   * Un grupo puede que no esté lleno.
   */
   private int numAlumnos;

   /**
   * Unico constructor de la clase.
   *
   * @param nombre Nombre que se desea asignar al grupo.
   * @param tamaño Tamaño con el que se crea el grupo.
   * @exception Exception Si el tamaño del grupo es menor que un
   * solo alumno.
   */
   public Grupo(String nombre, int tamaño) throws Exception {
      if (tamaño < 1)
         throw new Exception("Tamaño insuficiente");
      this.nombre = nombre;
      alumnos = new Alumno[tamaño];  // Se crea el grupo.
      numAlumnos = 0;  // Inicialmente hay cero alumnos.
   }
   /**
```

```
* Comprueba si el grupo está vacío. Es decir, si no tiene
* ningún alumno.
*
* @return  true si el grupo no tiene alumnos, false en caso contrario.
*/
public boolean estáVacío() {
   return numAlumnos == 0;
}

/**
* Comprueba si el grupo ya está lleno. Es decir, si no quedan
* sitios libres.
*
* @return  true si el grupo está lleno, false si hay huecos.
*/
public boolean estáLleno() {
   return numAlumnos == alumnos.length;
}

/**
* Dado un alumno lo añade al grupo.
*
* @param alumno Alumno que se desea añadir
* @exception Exception Si el grupo ya estaba lleno y no se puede
*           añadir.
*
*/
public void añadir(Alumno alumno) throws Exception {
   if (estáLleno())
      throw new Exception("Grupo lleno. Imposible añadir.");
   alumnos[numAlumnos] = alumno;
   numAlumnos++;
}

/**
* Elimina del Grupo un alumno que coincida con el alumno
* indicado. Tenga en cuenta que para ver si el alumno
* coincide con el indicado se utiliza el método equals
* definido en la clase Alumno, por lo que se considera que
* se ha encontrado si equals para los alumnos devuelve true.
*
* @param alumno Alumno que se desea eliminar.
* @return true si se ha eliminado un alumno, false si no se
*         ha eliminado ningún alumno.
* @see Alumno#equals(Alumno)
*/
public boolean eliminar(Alumno alumno) {
   int pos = buscar(alumno);
   if (pos < 0)
      return false;
   for (int i = pos; i < numAlumnos-1; i++) {
```

```
            alumnos[i] = alumnos[i + 1];
        }
        numAlumnos–;
        return true;
    }

    /**
     * Busca un alumno que coincida con los datos del
     * alumno dado dentro del grupo. Dos alumnos se supone
     * que coinciden si el método equals de los alumnos
     * devuelve true.
     *
     * @param alumno Alumno que se desea buscar.
     * @return Indice del Grupo donde se encuentra el alumno.
     *         Se devuelve -1 si el alumno no se encuentra en
     *         el grupo.
     * @see Alumno#equals(Alumno)
     */
    public int buscar(Alumno alumno) {
        for (int i = 0; i < numAlumnos; i++) {
            if (alumnos[i].equals(alumno))
                return i;
        }
        return -1;
    }

    /**
     * Imprime el nombre del grupo y los datos de
     * todos los alumnos que contiene utilizando
     * el método de imprimir alumnos definido en la
     * clase Alumno.
     *
     * @see Alumno#imprime
     */
    public void imprimir() {
        System.out.println(nombre);
        for (int i = 0; i < numAlumnos; i++) {
            alumnos[i].imprime();
            System.out.println();
        }
        System.out.println();
    }
}
```

Una vez generada la documentación con la herramienta javadoc o con alguna opción del entorno de desarrollo que esté utilizando podrá verla con cualquier navegador Web de la forma que se puede ver en el siguiente volcado de pantalla.

Package Class Use Tree Deprecated Index Help
PREV CLASS NEXT CLASS FRAMES NO FRAMES All Classes
SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

ejemplo1
Class Grupo

```
java.lang.Object
  └ejemplo1.Grupo
```

```
public class Grupo
extends java.lang.Object
```

La clase Grupo permite disponer de una reunión de alumnos de la clase Alumno. Todos los alumnos se guardan en un array cuyo tamaño se determina cuando se construye el objeto Grupo.

Version:
2.2, Enero de 2005
Author:
Jesús Sánchez Allende, Gabriel Huecas Fernández-Toribio, Baltasar Fernández Manjón, Pilar Moreno Díaz
See Also:
Alumno

Constructor Summary
`Grupo(java.lang.String nombre, int tamaño)` Único constructor de la clase.

Method Summary	
void	`añadir(Alumno alumno)` Dado un alumno lo añade al grupo.
int	`buscar(Alumno alumno)` Busca un alumno que coincida con los datos del alumno dado dentro del grupo.
boolean	`eliminar(Alumno alumno)` Elimina del Grupo un alumno que coincida con el alumno indicado.
boolean	`estáLleno()` Comprueba si el grupo ya está lleno.
boolean	`estáVacío()` Comprueba si el grupo está vacío.
void	`imprimir()` Imprime el nombre del grupo y los datos de todos los alumnos que contiene utilizando el método de imprimir alumnos definido en la clase Alumno.

Methods inherited from class java.lang.Object
`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructor Detail

Grupo

```
public Grupo(java.lang.String nombre,
             int tamaño)
      throws java.lang.Exception
```

Único constructor de la clase.

Parameters:
`nombre` - Nombre que se desea asignar al grupo.
`tamaño` - Tamaño con el que se crea el grupo.
Throws:
`java.lang.Exception` - Si el tamaño del grupo es menor que un solo alumno.

Method Detail

estáVacío

```
public boolean estáVacío()
```

Comprueba si el grupo está vacío. Es decir, si no tiene ningún alumno.

Returns:
true si el grupo no tiene alumnos, false en caso contrario.

---

estáLleno

```
public boolean estáLleno()
```

Comprueba si el grupo ya está lleno. Es decir, si no quedan sitios libres.

Returns:
true si el grupo está lleno, false si hay huecos.

---

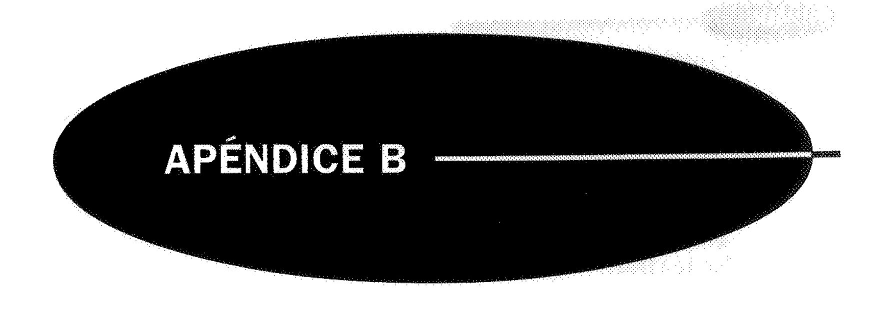
añadir

```
public void añadir(Alumno alumno)
            throws java.lang.Exception
```

Dado un alumno lo añade al grupo.

Parameters:
`alumno` - Alumno que se desea añadir
Throws:
`java.lang.Exception` - Si el grupo ya estaba lleno y no se puede añadir.

Figura A.1. Página de documentación de la clase Grupo en HTML.



Convenios de programación en Java

En este apéndice se incluye un conjunto de reglas que le permitirán seguir un estilo de programación uniforme para que sean fáciles de entender y corregir sus programas. Tener unas reglas para programar es importante porque:

- La mayor parte del tiempo se dedica a leer programas, por lo que resulta importante que sean fáciles de leer y de entender.
- Seguir unas reglas de programación permite mejorar la legibilidad del código escrito y entender fácilmente el código que no es de uno mismo.
- Si se da código propio a terceras personas, o se intenta explicar lo que se ha hecho, será más fácil de entender si se ha seguido un estilo consistente y similar, o igual, al de otros programadores.
- Casi nunca ocurre que todo el código que utiliza para un programa sea escrito por un solo programador, sino que se habrá desarrollado por un grupo de programadores.

Este apéndice se ha estructurado teniendo en cuenta los distintos elementos del lenguaje a considerar. Al principio pueden parecerle demasiadas cosas. Empiece por unas pocas y vaya leyendo nuevas guías de codificación según vaya ganando confianza con el lenguaje y conozca y aplique bien las anteriores.

B.1 ESTRUCTURA DE UN ARCHIVO FUENTE EN JAVA

Todo archivo fuente en Java tendrá la siguiente estructura:

- Comentario inicial: Cada archivo fuente contendrá un comentario inicial donde se describirá el objetivo de dicho archivo.
- Sentencia de paquete, si existe.
- Importación de las clases que se utilizan en esta clase.
- Declaración de clases e interfaces, que deberán aparecer en el siguiente orden:
  - Comentario de documentación de la clase o interfaz.
  - Sentencia de declaración de la clase o interfaz.
  - Atributos estáticos, primero los públicos, después los protegidos, después los de paquete y por último los privados.

- Resto de los atributos, primero los públicos, después los protegidos, después los de paquete y por último los privados.
- Constructores.
- Resto de los métodos. Se sugiere agrupar los métodos por funcionalidad, no por su accesibilidad (públicos, privados, etc.). De esta forma se trata de mejorar la legibilidad de los mismos y encontrarlos mejor dentro de la clase.

B.2 SANGRADO Y TAMAÑO DE LAS LÍNEAS

El sangrado deberá aplicarse a toda estructura que esté lógicamente contenida dentro de otra. El sangrado será de un tabulador. Se estima suficiente entre 2 y 4 espacios. Para alguien que empieza a programar suele ser preferible unos 4 espacios.

Las líneas no tendrán en ningún caso demasiados caracteres que impidan que se pueda leer en una pantalla. Un número máximo recomendable puede estar en torno a unos 70 caracteres, incluyendo los espacios de sangrado. Si una línea debe ocupar más caracteres debe dividirse en dos o más líneas. Para dividir una línea en varias, utilice los siguientes principios:

- Tras una coma.
- Antes de un operador, que pasará a la línea siguiente.
- Una construcción de alto nivel, por ejemplo una expresión con paréntesis.
- La nueva línea deberá alinearse con un sangrado lógico respecto al punto de ruptura.

Ejemplos:
Dividir tras una coma.

```
funcion (expresionMuyLargaDeExpresar1,
         expresionMuyLargaDeExpresar2,
         expresionMuyLargaDeExpresar3,
         expresionMuyLargaDeExpresar4);
```

Dividir tras una coma, alineando la siguiente línea de forma lógica.

```
var = funcion1(expresionMuyLargaDeExpresar1,
               funcion2(expresionMuyLargaDeExpresar2,
                        expresionMuyLargaDeExpresar3));
```

Mantener la expresión entre paréntesis en la misma línea.

```
nombreLargo1 = nombreLargo2 *
            (nombreLargo3 + nombreLargo4 - nombreLargo5) +
            4 * nombreLargo6;
```

Pequeña excepción: si la aplicación de estas reglas hace que el sangrado sea ilegible.

```
private static synchronized unNombreMuyLargo(int unArg,
        Object otroArg,
        String elTercerArg,
        Object unCuartoArg) {

if ((condición1 || condición2)
```

```
        && (condición5 || condicion6)
        && (condición7 || condición8)) {
    sentencias
}
```

B.3 COMENTARIOS

Los comentarios de un programa miden, de alguna forma, la calidad del código que se ha escrito. No sea parco utilizándolos.

Se debe poner un comentario:

- Al principio de cada archivo con la descripción y objetivo del mismo.
- Antes de cada método, explicando para qué sirve.
- Antes de cada algoritmo, explicando qué hace el algoritmo.
- Antes de cada definición de estructura de datos, indicando cuál es su objetivo.
- Antes de cada parte significativa del programa.

Los comentarios deben ir precedidos por una línea en blanco para separarlos lógicamente de la parte precedente del programa.

Existen varias formas de escribir comentarios:

Comentario multilínea: En este caso cada línea debe ir precedida por el carácter "*", excepto la primera y última que contendrán los símbolos de principio y fin de comentario, respectivamente.

```
/*
 * Esto es un comentario de un método
 * que sigue a continuación y donde se cuenta
 * para qué sirve dicho método.
 */
```

Los comentarios en una línea deben ir sangrados al mismo nivel que el código que comentan.

```
/* Se comprueba si funcion devuelve 3 o no */
if (funcion(unaVariable) == 3) {
    haceAlgo();
} else {
    haceOtraCosa();
}
```

Comentarios al final de la línea: Si se utilizan, deben ir suficientemente separados para distinguirlos como algo aparte del código.

```
if (funcion(unaVariable) == 3) {
    haceAlgo();        // si función devuelve 3
} else {
    haceOtraCosa();
}
```

Comentarios hasta final de línea: Anteponiendo el símbolo "//". No se recomiendan para comentar varias líneas seguidas. En el ejemplo se muestran dos formas de usar este comentario. Utilice una sola forma de manera consistente.

```
if (funcion(unaVariable) > 36) {
    // Hace algo con la variable dada
    haceAlgo(unaVariable);
} else {
    haceOtraCosa();        // Hace otra cosa
}
```

B.4 DECLARACIONES

Las declaraciones deberían hacerse una por línea, ya que facilita su documentación. Si alinea a la izquierda los nombres de las variables pueden resultar más fáciles de leer, aunque debe considerar cada caso.

```
int      pesoDeclarado;
int      pesoReal;
ListaMia unaLista;
```

Excepciones son: las declaraciones en la cláusula de inicialización de un for y las declaraciones de variables auxiliares:

```
int ancho, alto;
for (int i=0, j=1; ...
```

En ningún caso se utilizarán declaraciones en la misma línea de elementos que son de distinto tipo lógico, por ejemplo un entero y un array de enteros.

```
int numeroDeElementos, valoresReales[]; //EVITAR
int numeroDe Elementos;
int valoresReales[];
```

Aunque Java permite declarar variables en cualquier punto del código, hágalo preferiblemente sólo al principio de un bloque.

Evite declarar una variable en un bloque interno que oculte una declaración de un bloque exterior.

```
public int funcion (int a) {
    int var1;

    if (a > 2) {
        int var1;     // dar un nombre diferente
    }
}
```

B.5 ESPACIO EN BLANCO

El espacio en blanco no es un espacio desperdiciado. Resulta un espacio importante para separar unos elementos de otros de manera que sea fácilmente identificable lo que va junto y, por tanto, está relacionado y lo que se puede separar. Además de separar grupos de sentencias añada una línea en blanco:

- Separando dos métodos dentro de una clase.
- Entre la declaración de atributos de una clase y sus constructores y métodos.

- Entre la declaración de variables locales de un método y la parte de ejecución.
- Antes de un comentario dentro del código.

```
if (condicion) {
    sentencias
}

// Ahora se ve cuando se termina de repetirlo
while (otracondición) {
    sentencias
}
```

- Para separar código en trozos funcionales

```
if (n < 0) {
    throw new Exception("Error en entrada")
}

//cálculo de factorial
for (int i= 1; i < n; i++) {
    ...
}
```

En cuanto al uso de espacios en blanco, es mejor separar las operaciones que se están realizando para que queden claras:

Separar los operadores binarios con blancos.

```
a = b + c * variable.dameCoordenada();
```

Separar las partes de un for con un blanco delante.

```
for (int i=0; i < 23; i++) {
    sentencias
}
```

B.6 SENTENCIAS

Cada sentencia debe ir en una única línea.

```
i++;
j++;
```

Sentencia `if`:

```
if (condicion) {
    sentencias;
}

if (condicion) {
    sentencias;
```

```
} else {
    sentencias;
}

if (condicion) {
    sentencias;
} else if (condicion) {
    sentencias;
} else if (condicion) {
    sentencias;
}
```

Las sentencias dentro del `if` siempre deben ir entre llaves, aunque sólo contenga una sentencia. Si no se ponen las llaves puede provocar errores posteriores, tanto al añadir nuevo código como al interpretar lo que hace. De todas formas hay que considerar cada caso.

```
if (condicion)    // Evitar poner llaves
    sentencias;
```

Sentencia `switch`. Utilícela de la siguiente forma: Deje claras las opciones que no se interrumpen con `break`. Separe unos casos de otros lo suficiente. Deje claros cuáles son los distintos casos que se consideran en el `switch`. Toda sentencia `switch` debería tener el caso por defecto (`default`), aunque no haga nada. Ponga `break` en el `default`, aunque sea redundante.

```
switch (condicion) {
    case ABC:
        sentencias;
        /* sigue */
    case DEF:
        sentencias;
        break;

    case XYZ:
        sentencias;
        break;

    default:
        sentencias;
        break;
}
```

Sentencia `for`. Igual que en la sentencia `if` ponga siempre llaves en la parte que repite un `for` aunque sólo repita una sentencia.

```
for (inicializacion; condicion; continuacion) {
    sentencias;
}
```

En el caso del `for` para un conjunto de valores es similar.

```
for (variable : coleccion) {
```

```
        sentencias;
    }
```

Si el for no lleva ninguna sentencia indíquelo claramente poniendo el punto y coma en la línea siguiente.

```
    for (inicializacion; condicion; actualización)
        ;
```

Sentencia while. Lo que repite el while va entre llaves, aunque sólo sea una sentencia.

```
    while (condicion) {
        sentencias;
    }
```

Si while no repite ninguna sentencia utilice la siguiente forma:

```
    while (condicion)
        ;
```

La sentencia do-while siempre se escribe igual:

```
    do {
        sentencias
    } while (condicion);
```

Sentencia try - catch - finally. Utilícela de la siguiente forma:

```
    try {
        sentencias;
    } catch (Exception1 e) {
        sentencias;
    } catch (Exception2 e) {
        sentencias
    } finally {
        sentencias
    }
```

La sentencia return no debe pasar el valor que se devuelve entre paréntesis, para diferenciarlo de una llamada a un método.

```
    return valorCalculado;
```

B.7 ELECCIÓN DE NOMBRES

Utilice las siguientes reglas al asignar nombre a los distintos elementos de un programa:

- Clases: Utilice sustantivos, simples y descriptivos. Se deben escribir en minúsculas, uniendo las palabras que componen el nombre empezando cada palabra con mayúscula. Evite el uso de abreviaturas o acrónimos.

```
    class ObjetoComplejo ;
```

- Interfaces: Utilice adjetivos o sustantivos. Deben escribirse igual que las clases.

```
interface Dibujable ;
```

- Métodos: Utilice verbos, indicando la acción que realizan. Se escriben como las clases pero empezando con minúscula.

```
calculaCantidad();
extraeElemento();
```

- Atributos: Deben tener un nombre descriptivo que indique su utilidad. Evite el uso de nombres pequeños para los atributos, a excepción de los índices de bucles `for` donde se pueden utilizar `i`, `j`, etc. Si un atributo representa una colección de valores, debería ir en plural indicando este hecho.

```
int valorCalculado;
Lista elementosObtenidos;
```

- Constantes: Utilice nombres en mayúsculas, separados con guión bajo.

```
final int CAMBIO_MONEDA = 168;
```

- Enumerados: Utilice para los valores preferiblemente nombres en mayúsculas para distinguirlos de nombres de variables. Si son nombres comunes utilice la escritura habitual.

```
enum ColoresPrimarios = {ROJO, AMARILLO, AZUL}
enum DíasSemana = {Lunes, Martes, Miércoles, Jueves, Viernes, Sábado, Domingo}
```

B.8 PRÁCTICAS DE DISEÑO

En este apartado se dan consejos de diseño que debe aplicar en la programación de sus clases.

- Minimice la parte pública de una clase. Para ello defina primero la parte pública de una clase antes de empezar a codificar. Esta definición debería realizarse durante el diseño de la misma.
- Todos los atributos de una clase deberían declararse como privados en dicha clase. Para acceder a sus valores se deben de utilizar métodos de acceso, para obtener y fijar el valor de cada uno de dichos atributos.
- Métodos de acceso a los atributos de una clase:
  - Deberían utilizarse como la única forma de acceder a un atributo de una clase, incluso desde dentro de la misma clase, aunque pueda accederse directamente a ellos.
  - Para identificar los métodos de acceso se debe utilizar el prefijo `get` y `set`, para acceder y establecer el valor de una variable de la clase, respectivamente (en español se sugiere utilizar pon y dame o bien pon y el nombre del atributo). Si el tipo del atributo es `boolean` se puede poner el prefijo `is` (en español `es` o `está`). Por ejemplo:

```
getValorCalculado();
getElementosObenidos();
isCompleto();
ponValorCalculado();
dameNombre();
estaCompleto();
```

- Si un atributo contiene múltiples valores añada métodos para acceder, insertar y eliminar valores de dicho atributo de manera individual.
- Intente mantener la visibilidad de los métodos de acceso tan reducida como sea posible. Lo más aconsejable es que sean `protected`, permitiendo a las clases derivadas acceder a los mismos. Es común que los métodos de acceso para obtener un valor sean `public` y los métodos para fijar el valor sean `protected`.

- Los atributos calificados como `static` deben inicializarse siempre. No puede asegurar que no se accede a ellos antes de crear ningún objeto.